Level-based Indexing for Optimising XML Queries

Martin Francis O'Connor

Bachelor of Science in Computer Applications

A dissertation submitted in fulfilment of the

requirements for the award of

Masters of Science in Computer Applications

to the

Dublin City University

School of Computing

Supervisor: Dr. Mark Roantree

June, 2005

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Masters of Science in Computer Applications is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed _____

Student ID 53129172

Date June, 2005

Acknowledgments

I would like to thank my family, and in particular, my parents for their endless love, support and encouragement.

I would like to thank my supervisor, Mark Roantree, for giving me expert guidance and advice in the preparation of this thesis.
I would like to thank Zohra Bellahsène for her guidance and encouragement throughout the undertaking of my research.

I would like to thank all the members of ISG, including Dalen for instructing me in LaTeX, and my colleagues Noel, Seamus and Damir.

Last but most important of all, I want to thank God for His goodness and the many gifts He has given me, including life, love, faith and this opportunity of education.

Ad Majorem Dei Gloriam

Abstract

Many of the problems with native XML databases relate to query performance and subsequently, it can be difficult to convince traditional database users of the benefits of using semi- or unstructured databases. In particular, the ongoing development of the XQuery language requires that performance related issues are resolved. Presently, there still lacks an index structure providing efficient support for both navigational and structural queries and the traditional data-centric and content queries. This thesis presents a new extended index structure based on the preorder traversal rank and the level (or depth) rank of each node in a document tree. The extended index fully supports the navigation of all XPath axes while efficiently supporting data-centric queries. The ability to start path traversals from arbitrary nodes in a document tree also enables the extended index to support the evaluation of path traversals embedded in XQuery expressions. Furthermore, an encoding technique for this extended index structure is presented, whereby properties of a level ranking may be exploited to provide efficient and optimised path traversals and in certain cases, optimal solutions to path traversals.

Contents

List of Figures

Chapter 1

Introduction

The eXtensible Markup Language or XML [Wor04b] has been adopted as the new standard for data exchange on the World Wide Web and increasingly so in industry as the standard data interchange format for enterprise-wide application integration workflow systems. As more and more organisations and systems employ XML within their information management and exchange strategies, classical data management issues pertaining to XML's efficient and effective storage, retrieval, querying, indexing and manipulation arise. At the same time, previously uncharted information-modelling challenges appear [CRZ03].

XML first made its appearance on the international computing stage as a World Wide Web Consortium (W3C) working draft in November 1996 and was finally adopted as a recommendation in February 1998. It is a subset of the Standard Generalized Markup Language (SGML). Its initial goal was to enable generic SGML to be received and processed on the Web in a way that is now possible with HTML. For this reason, XML was designed for ease of implementation and for interoperability with both SGML and HTML. However, since 1998 the increasing importance and rate of adoption of XML within the wider computing domain has been considerable. The functionality, expressiveness and simplicity of XML have far exceeded its design goals, taking on a life and purpose of its own and thus, replacing and largely relegating SGML to specialised application domains.

In this chapter, the XML data model is introduced and the reasons for its rapid adoption are identified. An overview of the principle languages employed in the querying of XML and their importance in promoting the interoperability of XML is provided. The growth

in the development of new XML repositories and extensions to existing DBMS to facilitate the efficient storage, indexing and querying of XML data is examined in detail, with a view to identifying the open challenges still to be resolved. A number of these open challenges are then crystallised within a particular real-world context to provide a concrete motivation for the work presented in this research thesis.

The remainder of this chapter is structured as follows: the basic structure or datatype underlying the XML data model is presented in §1.1, in conjunction with the principle languages used to query XML data. In §1.2, the various types of XML databases that have evolved to date are outlined so as to identify the progress made in the XML repository domain and the various issues that remain outstanding. In §1.3, the Bologna Declaration is introduced so as to provide a context for the motivation of this research thesis. In §1.4, the specific issues that form the motivation of this research thesis are detailed. In §1.5, the contributions made in this research thesis are outlined and the conclusions are presented in §1.6.

1.1 The XML Data Model

XML provides an application-independent, language-independent and platform independent method to mark-up data. The primary purpose of mark-up data is to embed a description of the data in the data itself - that is embedding meta-data with data to provide semantic or contextual meaning, allowing the data to be interpreted. Thus, XML can be used to create abstractions of virtually all data structures and process states. Once captured in XML format, these data structures and process states can be queried. In addition, XML is standards-based. The W3C is the governing body that acts as the source for all specifications that govern the various XML initiatives. No major XML initiatives are based on proprietary protocols. All of the major players - Microsoft, Sun, IBM, HP and so on - adhere to the standards established by the W3C [MBC+04]. The universal independence of XML in conjunction with its semantic richness has positioned XML as the medium of choice for interoperability among distributed and heterogeneous computing systems.

The key ingredient to the successful adoption of XML is the expressive and extensible nature of XML. The basic structure underlying XML is the *tree*, which represents semi-structured data. Semi-structured data is such that the structure is not necessarily known in advance and is often self-describing as is the case with XML. Semi-structured data consists of an irregular and non-uniform organisation; it may have data with missing attributes and some attributes may be of different types within different data items. All of these variations are acceptable in XML documents. Thus, it may be seen that XML provides for an unlimited number for tree dialects, some of which have been formally described (structured) by Document Type Definition (DTDs) documents [BBC+98] or XML Schemas [Wor04c], while others are employed in an ad-hoc schema-less manner (semi-structured or unstructured). The database community is well advanced in adapting its technology to host large XML collections and to query these collections efficiently. It will be essential that these new technologies support the XML query language specifications such as XPath [Wor05b] and XQuery [Wor05a]. These specifications are key enablers in maintaining the interoperability among XML repositories.

XPath The XPath language enables access to individual parts of data elements in an XML document by the use of statements that express the path to the desired object [Kay04]. XPath models an XML document as a tree of nodes. There are various types of nodes, including element nodes, attribute nodes and text nodes. XPath operates on the abstract, logical structure of an XML document rather than its surface syntax. The primary purpose of XPath is to provide a means to enable the hierarchical addressing of nodes in an XML document tree, using XPath location steps. XPath is an expression language rather than a programming language. In its simplest form, an XPath expression takes an XML document as input and outputs a sequence of selected nodes that satisfy the expression. To navigate the tree of nodes in an XML document, XPath uses the concept of *axes*. XPath axes describe relationships between nodes in the document. The most commonly used axes are *child*, *ancestor*, *descendant*, *following* and *preceding*.

XQuery As XML has been adopted as the new standard for data exchange, it is natural that queries among applications and between databases should be expressed against data

in XML format. The SQL language, although robust and well-proven in the relational domain, is tree-unaware and not designed to exploit the hierarchical properties of XML encoded data. Subsequently, the SQL language is unable to meet the demands of an XML Query Language. The XQuery language specification is a response to this need. XQuery or XML Query Language is a W3C specification [Wor05a] designed to provide a flexible and standardised way of searching through (semi-structured) data that is either physically stored as XML or viewed as XML. The XQuery language is an extension of the XPath 2.0 language. XQuery can be used to query XML data that has no schema at all, or that is governed by a W3C XML Schema [Wor04c] or by a Document Type Definition (DTD) [BBC+98]. It is a functional language: instead of executing commands as procedural languages do, every query is an expression to be evaluated, and expressions can be combined quite flexibly with other expressions to form new expressions [CDF+04].

1.2 XML Databases

The rapid growth in the adoption of XML has highlighted the need for flexible and robust XML Database Management Systems (XML DBMS). Researchers from Industry and Academia have not been slow in responding to this challenge. A good number of XML DBMS are now available, each providing a range of functionality designed to address a particular requirement or application domain. The development of XML DBMS has evolved in two broad streams: *XML Enabled Databases* and *Native XML Databases.* A listings of over 30 Native XML Databases and 15 XML Enabled Databases is available at [Bou05]

XML Enabled Databases XML Enabled Databases are relational or object-relational databases that have been extended to store and process XML documents. Historically, their development sprung from the database community's view that XML was *yet another data format* to be catered for. XML Enabled Databases store XML using their own internal representation and provide a mapping mechanism to transform the data back into its original XML format. The XML documents are effectively shredded, decomposed and stored in rows and columns or as object-relational types within the database. This approach is suitable for highly structured data-centric applications that use XML data

primarily as a data interchange and transport format. An important consideration before deploying an XML Enabled Database is *Round Tripping*. Round Tripping describes the scenario whereby XML documents are stored in a database and then recreated through XML publishing. The ideal round tripping is achieved when the original and recreated XML documents are identical. This is not the case in many XML enabled systems due to the loss of some information like order or white spaces. For example, in [Ora03] Oracle's native CLOB storage can round-trip XML documents exactly, but object-relational storage can only round-trip at the level of the DOM [Wor04a].

Native XML Databases Native XML Databases are built from the ground up, with the goal of outperforming XML Enabled Databases by taking advantage of the inherent properties of the hierarchical structure of XML and implementing custom storage and indexing mechanisms to exploit them. An ideal Native XML Database should be based on the XML data model in which the XML document tree is the fundamental logical data unit [SBKJ02].

XML documents are characterised by both their content and their structure and thus, two major classes of queries are possible: query over content and query over structure. In [NLB+02], an analysis of several XML-Enabled DBMS and Native XML Databases were undertaken. The results confirmed that XML-Enabled relational and object-relational databases taking the data-centric view of XML data, process more efficiently the queries that manipulate data-centric documents, as opposed to document-centric data. On the other hand, Native XML databases are designed to handle raw XML data and documents and are more efficient in processing navigational or structural queries, showing poor performance for data-centric queries. Currently, no XML DBMS appears to offer the much sought-after balance between the data- and document-centric approaches. The choice of an XML DBMS to be adopted will be determined by the type of data to be stored and the class of queries to be performed.

To meet upcoming challenges, an XML DBMS will need to adopt the transaction management, concurrency control, security and administration features of the traditionally solid RDBMS, while performing both content and structural queries equally well.

1.3 Large-Scale XML Integration

On June 1999, twenty nine European governments signed the Bologna Declaration committing them to play their part in an action programme to create a European Higher Education Area (EHEA) by the year 2010 [Eur99]. The declaration recognises the value of coordinated reforms, compatible systems and common action while acknowledging the diversity of European higher education and reaffirming the independence and autonomy of individual universities. Ideally, when the Bologna Declaration has been implemented, a student will potentially construct the degree of their choice by selecting modules with the appropriate credits from various academic institutions spread throughout the European Union. To achieve these aims, a European framework supporting large scale data and resource integration is required to overcome the heterogeneous nature of countries and academic institutions.

The module specifications from all universities and educational institutions in Europe will constitute the basic units with which courses are constructed. A mediation service will be required to form the potentially large clusters of module data to facilitate the flexible course creation process. There has been much research in to the integration of XML data sources and the development of mediator architectures to facilitate an efficient query service [MP01] [ABFS02]. The result of the mediation process will be several large repositories of course data.

The XQuery for Large Scale Integration (XLIM) project [KR05], designed to look at some of the issues involved in the Bologna Declaration, extends the mediation architecture to provide query and meta-data services. In particular, with the onset of web services, a new class of XML documents have been created whereby some of the data is defined by means of embedded calls to web services [MAA+03]. The research presented in this thesis sets out to address the requirements of the query service. Such a query service should be efficient, scalable and reliable in supporting not only data and resource integration but also semantic interrogation. The universal nature and semantic richness of XML marks it out as the data format of choice for data and meta-data representation.

1.4 Motivation

A important component in the provision of an efficient, scalable and reliable query service is the underlying indexing service upon which it is built. As XML encoded data incorporates not only content but structure, the indexing service needs to capture this information and present it in a manner so as to be efficiently exploited. Moreover, unlike relational data, the meta-data is not stored in separate tables, but embedded throughout the data itself. Thus, it can be said that a query service operating on XML is reliant on a powerful indexing technique due to the semi-structured nature of the documents and the large data stores. This research thesis focuses on the provision of an indexing structure capable of supporting the demanding requirements of a query service for large scale XML repositories.

The operations and path traversals required in the querying of tree structured data present difficult challenges. There has been much activity on the specification and provision of extensions to the existing indexing mechanisms and processing models to enable the efficient exploitation of the structural properties of XML. The goal of this activity is to support, not only rapid navigational or structural queries but also efficient content-based queries [FK99] [ZND+01] [MWA+98]. There have also been several proposals [CSF+01] [LM01] [MS99] for new index structures to deal with these problems. However, virtually all of the proposals focus on support for step evaluation on the child and descendant-or-self axes, to the detriment of the remaining XPath axes. Many of the indexing structures often rely on query processing algorithms which call for implementation techniques that lie outside their natural domain. An example is the relational domain where such proposals incur associated drawbacks such as additional software layers and transactional and performance issues. Indeed, as trees in their abstract form may be queried using path expressions, the XPath language was defined to model and query an XML document as a tree of nodes. The XQuery specification moreover, facilitates embedded path traversals that may commence from any arbitrary node. Presently, there still lacks an index structure facilitating embedded XPath traversals from arbitrary nodes while providing at the same time, efficient XPath traversal evaluations incorporating both structural and navigational queries and the traditional content and data-centric queries.

1.5 Contribution

The contributions provided in this thesis may be summarised as follows:

- A new tree encoding mechanism is presented based solely on the preorder traversal rank and the level (or depth) rank of each node in the document tree. New conjunctive range predicates are defined based on the new tree encoding to support the evaluation of location steps on the principle XPath axes and proofs are provided to validate them. The new tree encoding mechanism and conjunctive range predicates combine to provide the foundation upon which the remaining benefits are based.

- An Extended Index structure (hereafter, referred to as the PreLevel structure) is then presented based on the new tree encoding that fully supports the evaluation of all XPath axes. Both the preorder traversal rank and level rank values may be determined during the initial parsing of the XML document and thus, the PreLevel structure has minimal computational overhead associated with its construction. The ability to start traversals from arbitrary context nodes in an XML tree also enables the PreLevel structure to support the evaluation of path traversals embedded in XQuery expressions. The tabular encoding of the PreLevel structure, being tree-aware, facilitates efficient structural and navigational queries in addition to content and data-centric queries. Furthermore, using our PreLevel structure, the properties of the level rank of a node may be exploited to provide optimised evaluations to various classes of XPath traversals.

1.6 Conclusions

In this chapter, the rise of the XML data format was introduced, its underlying data model outlined and the principle XML query languages presented. The evolution of XML databases to facilitate the rapid adoption of XML was detailed and the issues still outstanding in the provision of an adequate query service were explored. The Bologna Declaration was then presented to provide a context for this research work. In particular, the practical scenario and real world requirements motivating the XLIM query service were identified.

The significance and impact of the indexing service on an XML query service combined with the lack of an existing indexing mechanism to satisfy the XLIM query requirements, provided the impetus for this research work to develop a new indexing structure. The contributions of the new indexing structure were outlined in §1.5.

The thesis is organised as follows: chapter 2 reviews the existing *state of the art* indexing mechanisms for XML and identifies their strengths and weaknesses with respect to our requirements. Chapter 3 introduces the new tree encoding mechanism in addition to the new conjunctive range predicates that facilitate XPath axis navigation and the proofs of their derivation. Chapter 4 presents our new index: the PreLevel structure. The properties of the PreLevel structure are detailed and evaluations of a step location on the *ancestor*, *descendant*, *preceding* and *following* axes using our index are demonstrated. Chapter 5 outlines some of the classes of optimised XPath queries possible and specifically, how efficient level-based queries are possible. The conclusions are presented in chapter 6 together with proposed areas for future research.

Chapter 2

Related Research

In the previous chapter, we presented a motivation for an indexing structure to meet the requirements of the XLIM Query Service. In recent years and in line with the widespread adoption of XML, there has been much activity in extending and adapting existing indexing mechanisms. Furthermore, there has been a growth in the development of new index structures to fully exploit the structural and hierarchical properties of XML in addition to facilitating traditional data-centric and content-based queries. These developments, have focused on particular aspects of the indexing domain in an endeavour to overcome specific challenges. In particular, there are been several proposals for new index structures for XML [CSF+01] [LM01] [MS99]. With regard to their support for XPath, virtually all of these proposals focus on support for step evaluation on the child and descendant-or-self axes, with little or no focus on the remaining XPath axes.

In this chapter, several research projects covering existing *state-of-the-art* indexing structures and techniques are described. When examining these projects, the emphasis was on their ability to support efficient structural and navigational queries in addition to comprehensive support for the W3C XPath/XQuery specifications. This chapter is structured is follows: in §2.1 to §2.3 a discussion on a number of different research projects is provided; and in §2.4 some conclusions are presented.

2.1 The ORDPATH Research Project

On a conceptual level, XML documents consist of an ordered hierarchy of properly nested tagged elements. Elements can be labelled according to the structure of the document to facilitate query processing. Many labelling schemes have been proposed and a good overview of them is provided in [SHYY05]. Among the most popular and effective of them is an order-based labelling scheme that assigns a pair of numeric labels to each element based on the document order of its start and end tags. This labelling scheme lies at the center of many fundamental XML operations such as containment joins [ZND+01] and twig matching [BKS02], because it supports efficient evaluation of ancestor-descendant relationships among elements.

In [OOP+04] a hierarchical labelling scheme is introduced called ORDPATH that is implemented in the upcoming version of Microsoft® SQL Server™. This work was performed at Microsoft by the authors while on sabbatical from the University of Massachusetts at Boston. ORDPATH labels nodes of an XML tree without requiring a schema. A compressed binary representation of ORDPATH labels provide document order evaluation by simple byte-by-byte comparison and facilitates ancestor-descendant evaluations in a similar manner. In addition, ORDPATH is a dynamic labelling scheme supporting insertion of new nodes in arbitrary positions in the XML tree without the need to relabel any existing nodes.

2.1.1 ORDPATH Labelling Scheme

The ORDPATH Labelling scheme is conceptually similar to the Dewey Order encoding system described in [TVB+02]. Dewey Order is based on the Dewey Decimal Classification system developed for general knowledge classification. With Dewey Order, each node is assigned a vector that represents the path from the document's root to the node. Dewey order is "lossless" because each path uniquely identifies the absolute position of each node within the document.

XML data and a tree representing the XML hierarchy are shown in Example 2.1 and Figure 2.1 respectively, with a corresponding relational table containing the shredded

```
<BOOK ISBN="1-33463-812-3">
  <AUTHOR>
    <FIRST>Joe</FIRST>
    <LAST>Murphy</LAST>
  </AUTHOR>
  <SECTION>
    <TITLE>Rising Sun</TITLE>
    The sun rises
    <BOLD>every</BOLD> morning.
  </SECTION>
</BOOK>
```

Example 2.1: Sample XML data

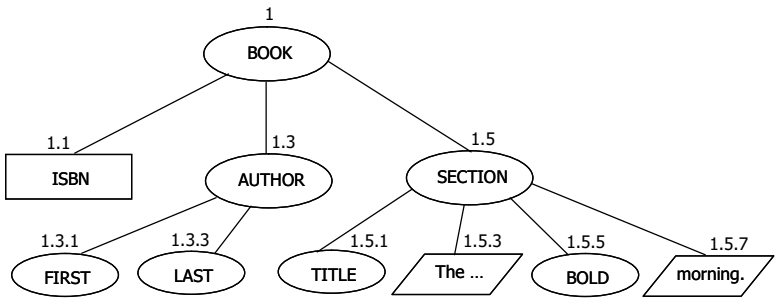


Figure 2.1: XML tree for XML data of Example 2.1

XML node data shown in Figure 2.2. This relational table is referred to as the NODE table. When the XML document is initially loaded and parsed, each node in the XML tree is traversed in document order, the ORDPATH labels are generated and the NODE table is populated. In the ORDPATH values of Figure 2.2 (such as "1.5.3"), each dot separated component value ("1", "5", "3") reflects a numbered tree edge as successive levels down the path from the root (itself having a 0-length ORDPATH) to the node represented. Note that only positive, odd integers are assigned during the initial load; even-numbered and negative integer component values are reserved for later insertion into an existing tree. ORDPATH values are not stored as a dotted-decimal string but rather in a compressed binary representation. The NODE TYPE column of Figure 2.2 contains coded values for various node types: 1 for an element, 2 for an attribute and so on. The TAG column contains coded tags. The VALUE column contains variable-type data that is associated with some nodes.

ORDPATH	TAG	NODE TYPE	VALUE
1.	1 (BOOK)	1 (Element)	null
1.1	2 (ISBN)	2 (Attribute)	'1-33463-812-3'
1.3	3 (AUTHOR)	1 (Element)	null
1.3.1	4 (FIRST)	1 (Element)	'Joe'
1.3.3	5 (LAST)	1 (Element)	'Murphy'
1.5	6 (SECTION)	1 (Element)	null
1.5.1	7 (TITLE)	1 (Element)	'Rising Sun'
1.5.3	--	4 (Value)	'The sun rises'
1.5.5	8 (BOLD)	1 (Element)	'every'
1.5.7	--	4 (Value)	'morning.'

Figure 2.2: XML Shredded into Relational "NODE" Table.

Primary Index. An ORDPATH primary key (with a cluster index) on the NODE table provides efficient query access to XML data. For example, a query that retrieves all descendants of x will find them clustered on disk just after x, in document order, so retrieval is optimal.

Secondary Indexes. There are two principle secondary indexes employed by the ORDPATH labelling scheme. The Element and Attribute TAG (with integer id) index supports fast lookup of elements and attributes by name. The Element and Attribute VALUE index supports lookup of the variable-type value data.

2.1.2 ORDPATH Query Plans

The following example is used to illustrate an XPath ancestor-descendant query:

1. *//Book//Title[. = "Red Rose"]*

In general, descendant connections between node sets that are independently described may be treated as joins. If the *Book* node has many descendants (a valid assumption for any indexing mechanism claiming to support scalability), the ORDPATH approach is to separately locate the sequence of Book elements and the sequence of Title elements that have "Red Rose" as a value (using the VALUE secondary index), then merge join the two sequences. Both nodes sequences will be in increasing order of ORDPATH values,

since each of them is located by a single value in the primary index; furthermore, a merge join of ancestor-descendant node sequences may be treated in much the same way as an equal-match join as detailed in [ZND+01].

The principle features of ORDPATH are its support for dynamic updates and insertions for both schema-based and schema-less XML documents. The compressed binary representation of ORDPATH labels, enable efficient evaluations of location steps on the ancestor-descendant axis. Furthermore, the prefix-free property of ORDPATH labels (described in [OOP+04]) enable the efficient evaluation of the exact nature of the ancestor-descendant relationship (i.e. parent, grandparent, etc).

However, in order to facilitate querying on all XPath axes of hierarchy and precedence, including axes such as siblings, a new secondary index based on the LEVEL (or node depth in the document tree) is required. This LEVEL Index, in conjunction with the aforementioned primary and secondary indexes, enables ORDPATH to support the evaluation of *all* XPath axes.

2.1.3 Limitations

Although the principle advantage of the ORDPATH labelling scheme over existing mechanisms is its support for updates and insertion of nodes, there are still some issues to be addressed. ORDPATH's support for full XPath axes evaluation necessitates the overhead of a second merge join between the LEVEL Index and the ORDPATH primary Index. The experiments performed in [ZND+01] demonstrated the significant impact of, not only the number of joins, but the actual join algorithms employed, have on query performance. The requirements of two merge joins make a significant impact on the query performance over large document collections.

2.2 Indexing In The eXist Database

eXist is an open-source native XML database and provides for the schema-less storage of XML documents in hierarchical collections [Mei02]. The database is completely written in Java and may be deployed in several ways; running standalone, inside a servlet engine or

directly embedded in an application. Although a lightweight database, the eXist query engine implements index-based XPath and XQuery processors, using indexes for all element, text and attribute nodes. Based on path join algorithms, a wide range of path expression queries is processed using only indexed information. eXist will not load the actual nodes unless it is required to do so, for example, to display the results.

This section will introduce the indexing system implemented in eXist, detail the numbering scheme used at the core of the database and lastly highlight some limitations that rule out its suitability as an indexing mechanism for our XLIM query service.

2.2.1 Overview

The indexing scheme employed by eXist has been inspired by three contributions from recent research [ZND+01] [LM01] [LYYB96]. eXist indexing system uses a numbering scheme to identify XML nodes and determine relationships between nodes in the document tree. A numbering scheme assigns a unique identifier to each node in the document tree by traversing the tree in preorder or level order. The generated identifiers are then used in indexes as a reference to the actual nodes. A numbering scheme should provide a mechanism to quickly determine the structural relationships between a pair of nodes and also all nodes in a document that satisfy a particular relationship.

In [ZND+01], a numbering scheme is proposed whereby each node in the document tree may be identified by its document id, its position and its nesting depth within the document. An element is identified by a 3-tuple identifier *(document id, start:end, nesting level)*. The position *start:end* can be generated by counting word numbers in the document. Using the 3-tuple, ancestor-descendant relationships can be determined between a pair of nodes by the proposition: A node x with 3-tuple identifier (D_1, S_1:E_1, L_1,) is a descendant of a node y with the 3-tuple (D_2, S_2:E_2, L_2) if and only if $D_1 = D_2$; $S_1 < S_2$, and $E_2 < E_1$. However, the 3-tuple identifiers generated by this numbering scheme consume a lot space in the index and significantly impact on query performance when compared to single tuple identifiers generated by preorder and level-order traversals.

In [LYYB96], a numbering scheme is proposed which models the document tree as a complete k-ary tree, where k is equal to the maximum number of child nodes of an element

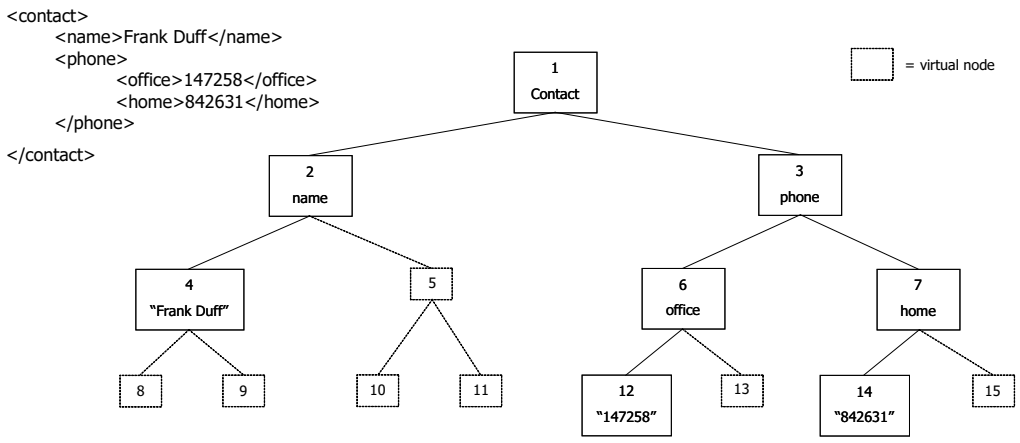


Figure 2.3: A sample XML document modelled as a complete 2-ary tree using level-ordering.

in the tree. Thus, every non-leaf node in the tree has the same number of children. A unique identifier generated from a level-order traversal is assigned to each node in the tree. A level-order traversal is such that when a tree is displayed, it is visited from top to bottom going from left to right. For any node having less than k children, virtual child nodes are inserted to fill the gaps. Figure 2.3 shows the level-order identifiers assigned to the nodes of a simple XML document which is modelled as a complete 2-ary tree.

This numbering scheme has several important properties: from a given identifier, one may easily determine the identifier of its parent, sibling and (possibly virtual) child nodes.

However, the existence of virtual nodes, required to balance the tree into a complete k-ary tree means that some node identifiers are wasted. Furthermore, the completeness constraint imposes a major restriction on the maximum size of a document to be indexed. In practise, many documents contain more nodes in some distinct subtree of the document that in others [CRZ03]. For example, a typical article will have a limited number of top-level elements like chapters and sections, which the majority of nodes consists of paragraphs and text nodes located below these top-level elements. In a worst case scenario, in which a single node at an arbitrarily deep level of the document node hierarchy has the largest number of child nodes, a large number of virtual children must be inserted at all tree levels to satisfy the completeness constraint, so the assigned identifiers grow quickly even for small documents.

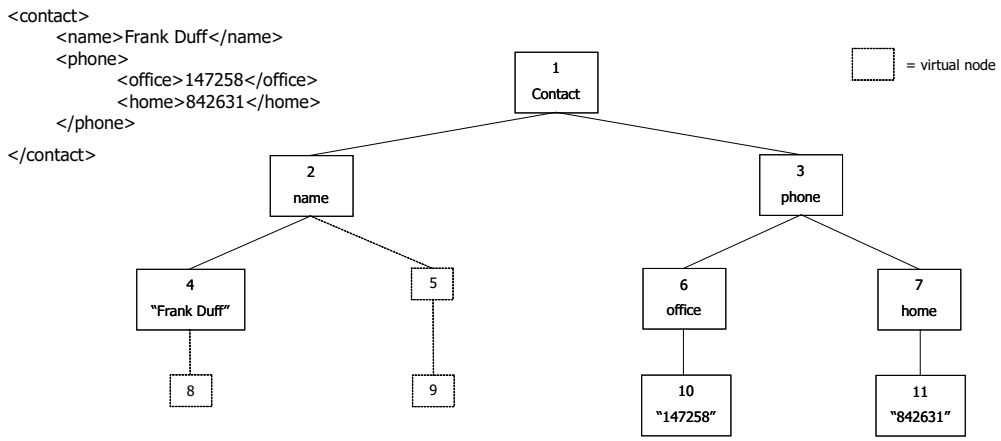


Figure 2.4: A sample XML document with unique identifiers generated by eXist.

2.2.2 eXist Numbering Scheme

The numbering scheme implemented in eXist overcomes the document size limitation by partially dropping the completeness constraint in favour of an alternative scheme. The document is no longer viewed as a complete k-ary tree. Instead the number of children each node may have is recomputed for every level of the tree, such that: for two nodes x and y of a document tree, *size(x)* = *size(y)* if *level(x)* = *level(y)*, where *size(n)* is the number of children of node n and *level(m)* is the length of the path from the root node of the tree to node m. The additional information on the number of children a node may have at each level of the document tree is stored with the document in a simple array. Figure 2.4 shows the unique identifiers generated by eXist for the same document depicted in Figure 2.3.

Benefits

eXist's numbering scheme does not effect the general properties of the assigned level-order identifiers. From a given unique identifier, one may still compute the parent, sibling and child node identifiers using the additional information on the number of children each node may have at every level of the tree. The advantage of eXist's numbering scheme is that it takes into account that typical XML documents will have a larger number of nodes at some lower level of the document tree while there are fewer elements at the top level of the hierarchy. Changing k at a deeper level of the document tree has no effect on nodes

at higher levels of the tree. The document size limit is raised considerably to enable the indexing of much larger documents.

2.2.3 eXist Query Plans

In order to evaluate path expressions, the eXist query processor decomposes a given path expression into its component steps. For example, to process the XPath expression, */play//speech*, the query processor will load the root element *play* for all documents in the input set. Secondly, the set of *speech* elements is retrieved for all input documents. Now, we have two node sets containing potential ancestors and descendants for each of the documents in question. Each node set consists of <*document-id, node-id*> pairs, ordered initially by document identifier and then by unique node identifier.

To find all *speech* nodes that are a descendant of the *play* nodes, an ancestor-descendant path-join algorithm is applied to the two sets. eXist's path join algorithms are similar to those presented in [LM01]. The ancestor-descendant path join algorithm recursively replaces all node identifiers in the descendant set with their parent's node id and loops through the two sets to find equal pairs of nodes. If a matching pair is found, it is added to the resulting node set. Thus, eXist does not need to access the DOM nodes in the XML store to evaluate the expression /play//speech. The hierarchical structure of path expressions is processed entirely using the numeric identifiers provided in the index.

2.2.4 Limitations

Although the eXist query processor can evaluate structural relationships between nodes in the document tree using index-based processing alone, the path-join algorithms used to determine structural relationships are based on level-order traversals and are thus, inefficient in comparison to other numbering schemes such as preorder traversal. For example, in the ancestor-descendant path-join algorithm described in §2.2.3, every node (or its ancestors) in the descendant set must be compared with every node in the ancestor set, possibly several times depending on the depth of the descendant in the document tree, to determine the result set. This is effectively a worst case scenario. An indexing

structure is required that will reduce to a minimum the number of comparison required when determining structural relationships between nodes.

2.3 XPath Accelerator Index Structure

In [Gru02], the author proposes a database index structure called the *XPath Accelerator*, which has been specifically designed to support the evaluation of XPath queries. The index structure uses the notion of document regions created by the XPath axes in conjunction with preorder and postorder traversals to encode XML documents. XPath document regions and the properties of tree traversal are discussed in more detail in chapter 3 as part of a detailed discussion of tree traversals.

2.3.1 Overview

Evaluating a location step on a major XPath axis (*ancestor*, *descendant*, *preceding* and *following*) amounts to a rectangular region query in the *pre/post* encoded plane. To support the remaining XPath axes and nodes tests, extra information must be stored. For a context node v, axes *ancestor-or-self* and *descendant-or-self* simply add v to the ancestor or descendant regions respectively. To support the *following-sibling* and *preceding-sibling* axes, the parent's preorder rank *par(v)* of each node v is recorded because siblings shared the same parent. *par(v)* characterises the *child* and *parent* axis also. To support the *attribute* axis and in line with XPath semantics, to exclude the attribute nodes from all other axes, a boolean attribute *att(v)* is maintained for each node v. Finally, name tests are supported by attribute *tag(v)* which stores the element tag or attribute name for node v.

This completes the encoding used by the XPath Accelerator. Each node v is represented by its 5-dimensional description:

- *desc(v)* = {*pre(v), post(v), par(v), att(v), tag(v)*}

An XPath axis corresponds to a specific query window in the space of node descriptors.

Axis α	Query window *window(α,v)*				
	pre	post	par	att	tag
child	(pre(v),∞)	[0,post(v))	pre(v)	false	*
descendant	(pre(v),∞)	[0,post(v))	*	false	*
descendant-or-self	[pre(v),∞)	[0,post(v)]	*	false	*
parent	[par(v),par(v)]	(post(v),∞)	*	false	*
ancestor	[0,pre(v))	(post(v),∞)	*	false	*
ancestor-or-self	[0,pre(v)]	[post(v),∞)	*	false	*
following	(pre(v),∞)	(post(v),∞)	*	false	*
preceding	(0,pre(v))	(0,post(v))	*	false	*
following-sibling	(pre(v),∞)	(post(v),∞)	par(v)	false	*
preceding-sibling	(0,pre(v))	(0,post(v))	par(v)	false	*
attribute	(pre(v),∞)	[0,post(v))	pre(v)	true	*

Figure 2.5: XPath axes α and their corresponding query windows *window(α,v)* (context node v)

Figure 2.5 summarises the windows together with the corresponding axes they implement. A node *v'* is inside the query window if its descriptor *desc(v')* matches the query window component by component (for the first two components, *pre(v')* and *post(v')* must lie inside the respective ranges). A star entry (*) indicates a *do not care* match which always succeeds. The query window for the name test *α::n* is *window(α,v)* with its *tag* entry set to *n*.

2.3.2 Benefits and limitations of the XPath Accelerator

The XPath Accelerator is capable of supporting the evaluation of *all* XPath axes. The index maintains document order among nodes and supports XPath traversals beginning for arbitrary context nodes. Furthermore, the ability to start traversals from arbitrary context nodes in the XML document enables the index to support XPath expressions embedded in XQuery statements. The XPath Accelerator is a relational storage structure in that the index can be constructed and queried using relational idioms only. Its implementation can benefit from the advanced indexing technology currently available in RDBMSs.

A Stretched Pre/Post Plane

In [GVT04] several shortcomings of the XPath Accelerator are noted and enhancements to overcome these shortcomings are proposed. These enhancements however, come with various costs that impact on query performance. One such enhancement is the *Stretched pre/post plane*.

All axes query windows in the two-dimensional *pre/post* plane depend on a range selection in the *pre* as well as the *post* dimension. As nodes in these query windows are determined using two independent range queries in both the *pre* and the *post* dimensions, the larger the query window, the greater the number of false hits encountered. These nodes must be filtered during a subsequent intersection. An enhancement proposed to the *pre/post* encoding, namely the Stretched *pre/post* plane, requires a modification to the construction of the *pre/post* plane to facilitate a location step on the descendant axis with a single range scan over either the *pre* or the *post* ranks. The advantage of this enhancement is that the query window used to evaluate a descendant step in the stretched *pre/post* plane never encounters false hits. However, the modification to the construction of the *pre/post* plane, results in a stretching of the plane whereby the preorder ranks are no longer consecutive and therefore, the pre-column is no longer dense. Consequently, preorder ranks must be indexed separately, requiring more bytes per node. This has immediate consequences on query performance and on the volume of data to be accessed. Also, for any implementation using a fixed-bit width representation for the coupled *pre/post* ranks, the stretching of the *pre/post* plane implies the number of nodes that may be represented is effectively halved, when compared with the non-stretched case.

Jungle Storage Manager Implementation Experience

In [VFS04], the experience of building Jungle, a secondary storage manager for Galax, an open source implementation of the family of XQuery 1.0 specifications is presented. They chose to implement the Jungle XML indexes using the XPath Accelerator. However, one significant limitation they encountered was the evaluation of the child axis, which they found to be as expensive as evaluating the descendant axis. They deemed this limitation

to be unacceptable and designed their own alternative indexes to support the child axis. Furthermore, the Jungle implementation experience also highlighted the significant overhead imposed at document loading time by a postorder traversal, a necessary component in the construction of the XPath Accelerator.

2.3.3 Modified XPath Accelerator Encoding

In [GST04], the authors of the XPath Accelerator proposed a modified encoding for their index structure where the postorder rank is no longer used. The tree encoding is extended for a node v by:

1. *v.size*: the number of nodes in the subtree below v.
2. *v.level*: the length of the path from the root node of the document tree to node v.

The new *pre/size/level* encoding supports all XPath axes and provides the same functionality as the original encoding. This is made possible by the fundamental property describing the relationships between nodes in a tree, namely: for any node v in a tree t,

- *pre(v) + size(v) - level(v) = post(v)*

The modified encoding brings several advantages over the original encoding: the evaluation of a location step on the child axis is more efficient, the document loading times should decrease significantly now that postorder traversals are no longer required and the stretched *pre/post* plane is no longer necessary as all queries are now performed in the preorder dimension.

Nevertheless, several shortcomings may still be identified. The evaluation of *all* members of several XPath axes, such as the *child*, *following-sibling* and *preceding-sibling* axes are inefficient. The inefficiency lies in the fact that almost all descendants of the given context node need to be processed in order to identify the members of the respective axis. For documents with a small number of nodes at the higher levels and large numbers of nodes at the lower level, a typical scenario as identified in [MBV03], a significant amount of processing will be spent on examining nodes that are not members of the required axis.

2.4 Conclusions

In recent years there has been much research into the provision of indexing structures to support the storage, retrieval and querying of XML documents in large XML repositories. Many research projects extended existing indexing mechanisms taken from the relational and object-relational domain to make them *tree-aware.* Other research projects developed new indexing structures from the ground up, designed to harness the intrinsic properties of tree-structured data. In this chapter we have examined a number of *state-of-the-art* indexing structures from current research, provided an overview of their functionality and highlighted their benefits and limitations.

In the ORDPATH project, a hierarchical labelling scheme derived from the Dewey Order encoding system was presented. ORDPATH is a dynamic labelling scheme supporting insertions of new nodes in arbitrary positions in a (schema-less) XML document without the need to relabel existing nodes. A compressed binary representation of ORDPATH labels provide document order evaluation by simple byte-by-byte comparison and facilitates ancestor-descendant step evaluations in a similar manner. Yet, the support for dynamic updates comes with the price of high computational costs in query performance. In particular, when evaluating a step location on a level-based axis, such as the following-sibling and preceding-sibling axes, two merge joins are required. The evaluation of *all* members of an XPath axis, likewise requires two merge joins.

In the eXist database project, the indexing mechanism for eXist, an open-source native XML database was presented. Like ORDPATH, eXist supports the indexing of schema-less documents. The index mechanism used in eXist has been derived from the properties of three related research projects. In [ZND$^+$01], a 3-tuple numbering scheme is proposed that can determine ancestor-descendant relationships using simple evaluations. However, the 3-tuple identifier is quite large and significantly impacts on the query performance over large collections. In [LYYB96], a numbering scheme which models an XML tree as a complete k-ary tree is presented. However, the completeness constraint of this encoding mechanism severely limits the size of the documents that may be indexed. eXist adapts this numbering scheme and extends it by partially relaxing the completeness constraint to

allow the number of children to be recomputed for each level in the document tree. The advantage of eXist's alternative numbering scheme is the support for the indexing of larger document sizes. eXist indexing mechanism supports all XPath axes evaluations (excluding following-sibling and preceding-sibling axes) and maintains indexes on all elements, text and attribute node and thus supports structural query evaluation using information contained in the indexes alone. However, the path join algorithms, derived primarily from [LM01], are necessarily based on the level-order traversal ranks and thus, are inefficient in comparison to the preorder-based numbering schemes.

Lastly, the XPath Accelerator index structure was presented. This index structure is based on a preorder numbering scheme and most closely meets the goals of the XLIM Query Service. It has been specifically designed to support the evaluation of all XPath axes. The index maintains document node order and supports XPath traversals beginning from arbitrary context nodes and thus, supports evaluation of path expressions embedded in XQuery statements. The original XPath Accelerator *pre/post* encoding had several shortcomings as outlined in §2.3.2 and some of those have been overcome with the adoption of the new *pre/level/size* encoding. Although the XPath Accelerator efficiently evaluates *all* members of the major XPath axes, evaluations of all members on several of the remaining XPath axes are inefficient. Furthermore, evaluations of several classes of queries that are not XPath axes based, but nonetheless are common occurrences, are badly supported. An example of such a query is the selection of all grandchildren of a context node.

In this chapter, several major research projects incorporating existing *state-of-the-art* indexing structures were examined, their benefits appraised and their limitations identified. During the study of these projects, some key characteristics emerged, which together with our analysis in chapter 1, provide the functional requirements for a suitable XML indexing structure for large XML repositories. These requirements may be enumerated as follows:

1. The index structure fully supports the XPath and XQuery data model.
2. The index structure provides comprehensive support for and efficient evaluations of hierarchical and structural queries.
3. The index structure facilitates scalability, not only in terms of document size but

also in terms of the size of the XML data repositories that may be indexed.

4. The index structure will be generic in so far as it will not specify a specific storage model for the documents nor an implementation model for its operation. This requirement ensures the index structure may be deployed in various operating scenarios.

At this point, clear requirements for an new indexing structure have been identified. However, before the new index structure is introduced in chapter 4, it is necessary to introduce the new encoding mechanism (in chapter 3) and its properties upon which the new index structure is built.

Chapter 3

PreLevel Encoding

In chapter 1, the fundamental datatype or structure underlying the XML data model was identified as a tree. Tree structures are well researched and understood in computing research and have presented many challenges to researchers in the areas of query processing and information retrieval. In this chapter, a review of the properties of tree traversal is provided and its relationship to the XPath language specification is explored. A new tree encoding mechanism based on the preorder traversal rank and level rank of a node is then presented. The new tree encoding constitutes the cornerstone of the new indexing structure to be presented in chapter 4 and addresses the requirements identified in chapter 2. For each of the principle XPath axes, new conjunctive range predicates for performing a location step on the axis are defined and their corresponding proofs provided. The new conjunctive range predicates have been derived from the intrinsic properties of the preorder traversal ranks and level ranks alone.

The chapter is structured as follows: In §3.1, the XPath working draft and its associated *partition* property is reviewed. In §3.2, the core properties of tree traversal are examined and specifically, how they relate to XML document encoding, concluding with a numbering scheme exploiting the properties of tree traversal. In §3.3, our new tree encoding mechanism is introduced and the new conjunctive range predicates defining location steps on the principle XPath axes together with proofs of their derivation are presented. Finally, §3.4 concludes the chapter.

3.1 The XPATH Language Specification

The World Wide Web Consortium (W3C) standard for uniquely addressing a node in an XML tree is the XPath Language 2.0 working draft recommendation [Wor05b] (hereafter referred to as the XPath specification). XPath models an XML document as a tree of nodes. There are various types of nodes including element nodes, attribute nodes and text nodes. XPath provides a means of hierarchical addressing of nodes in an XML tree using XPath *location steps*. An XPath location step consists of three parts: an optional *axis* which selects a set of candidate nodes; a *node test* which specifies the node name and the type of node to be selected, and finally zero or more *predicates* which further filter the nodes according to arbitrary selection criteria.

A sample XPath location step is as follows: *child::title[position()=1]*. The *child* is the name of the axis; the *title* is the node test and *[position()=1]* is the predicate. An abbreviated syntax is described in the XPath specification whereby the axis may be omitted and a shorthand notation be used instead.

The axis returns a set of nodes relative to a specific context node, for example, its ancestors or descendants. According to the XPath specification, there are 11 axes available to describe the various unique relationships that may exist between nodes (ignoring namespace and attribute nodes). However, there are four XPath axes that are of particular interest: the **descendant**, **ancestor**, **preceding** and **following** axes. Hereafter, these axes shall be referred to as the *primary* axes. The given context node constitutes the **self** axis. The remaining XPath axes (**parent**, **child**, **descendant-or-self**, **ancestor-or-self**, **following-sibling** and **preceding-sibling**) determine either supersets or subsets of one of the primary axes and may be evaluated from them.

XPath Partition Property The XPath specification has been defined with the following special property: the **ancestor**, **descendant**, **preceding**, **following** and **self** axes partition an XML document as disjoint spaces and together they contain all nodes in the XML document. Thus, as a given context node resides in the **self** axis, all other nodes in the XML document fall into one of the four *primary* partitions. This special property is significant and is the foundation upon which the *XPath Accelerator* proposed in [Gru02]

is built. The PreLevel encoding, to be presented in §3.3, is an extension to the *XPath Accelerator* and exploits the XPath partition property.

3.2 Tree Traversal

In this section, the core properties of tree traversal are reviewed and in particular, how they relate to the processing of XML documents. The properties of tree traversal constitute an essential component of the new indexing structures and algorithms presented in this thesis. Finally, a node numbering mechanism using tree traversal is presented.

3.2.1 Overview

Tree traversal is the process of visiting each node in a tree data structure. Tree traversal provides for sequential processing of each node in what is, by nature, a non-sequential data structure (e.g., semi-structured data). Such traversals are characterised by the order in which the nodes are visited. Thus, given a tree node structure which contains a node *value* and references to its two children *left* and *right*, the recursive function illustrated in Example 3.1 describes a preorder traversal.

```
Visit(node)
  Print node.value
  if node.left != null then Visit(node.left)
  if node.right != null then Visit(node.right)
```

Example 3.1: A recursive Preorder Traversal function.

In preorder traversal, each node v is visited and assigned its preorder traversal rank *pre(v)* before its children are recursively traversed left to right. It is worth observing at this point that the act of parsing an XML document in document order, that is, processing each line from left to right and from top to bottom, corresponds to a preorder traversal of the XML document tree. Thus, a preorder traversal maintains the document node order of an XML document, an important pre-requisite for any XPath evaluations.

In a postorder traversal, a node v is assigned its postorder traversal rank *post(v)* after all its children have been traversed from left to right. If the PRINT statement in the

recursive function given in Example 3.1 was placed at the end, the function would describe a postorder traversal.

In summary, a tree traversal encoding of an XML document has the following three properties:

1. Rooted: The XML document always has one and only one root node.
2. Ordered: The nodes in an XML document are ordered in a defined way.
3. Labelled: Each node is assigned a unique label or traversal rank.

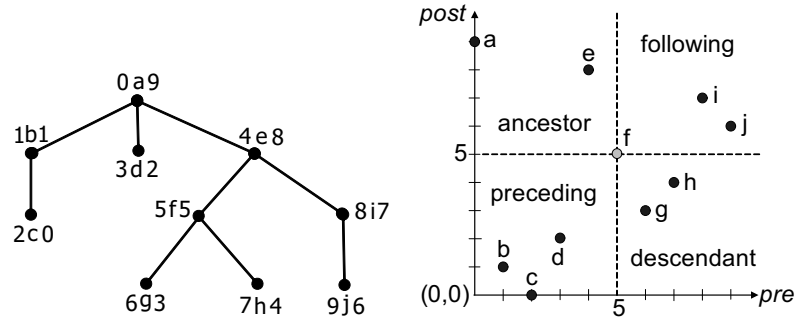
3.2.2 Numbering Nodes

Dietz's numbering scheme presented in [Die82] uses tree traversal order to determine the **ancestor-descendant** relationship between any pair of nodes in a tree. His proposition is: for two given nodes x and y of a tree T, x is an **ancestor** of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal. Earlier in §3.1, the special *partition* property of XPath semantics was introduced. This property is preserved by the *pre/post* encoding and extends Dietz's numbering scheme to incorporate all XPath axes. If the encoded nodes are mapped to the *pre/post* cartesian plane, then for any given context node c, the four *primary* XPath axes coincide with the disjoint rectangular regions partitioning the plane, the point of intersection being the context node c (the **self** axis) as illustrated in Figure 3.1(b).

The XPath specification defines a *relative location path* as consisting of a sequence of one or more location steps separated by a "/" . In the cartesian plane, a location step on the primary axes may be evaluated using the simple conjunctive range predicates given in Example 3.2.

```
v ∈ c/descendant   ⇔  pre(v) > pre(c) ∧ post(v) < post(c)      (i)
v ∈ c/ancestor     ⇔  pre(v) < pre(c) ∧ post(v) > post(c)      (ii)
v ∈ c/following    ⇔  pre(v) > pre(c) ∧ post(v) > post(c)      (iii)
v ∈ c/preceding    ⇔  pre(v) < pre(c) ∧ post(v) < post(c)      (iv)
```

Example 3.2: Conjunctive range predicates defining location steps in the pre/post plane.



(a) XML skeleton tree with pre/post rank.

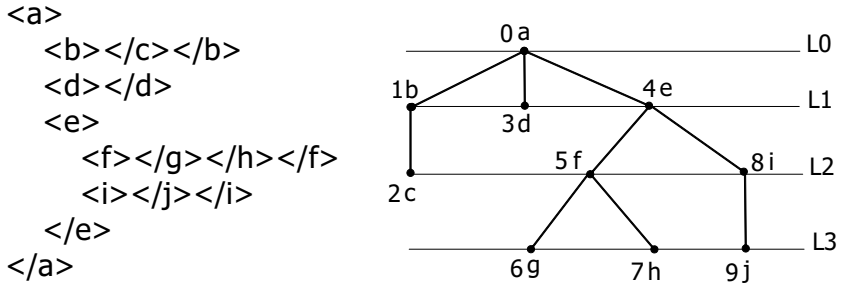
(b) pre/post carterian plane with context node f.

Figure 3.1: A sample XML tree and the *pre/post* cartesian plane.

The first conjunctive range predicate states an arbitrary node v is a **descendant** of the given context node c if and only if the preorder rank of v is greater than the preorder rank of c and that the postorder rank of v is less than the postorder rank of c. The conjunctive range predicates in Example 3.2 are the theoretical foundation of the *XPath Accelerator*. The contribution of the *pre/post* encoding mechanism to the indexing of XML documents in addition to its limitations have already been discussed in detail in chapter 2. In the next section, an extension to the preorder traversal encoding mechanism is presented which provides all of the significant benefits of the combined *pre/post* encoding. Furthermore, the intrinsic properties of this new encoding may be exploited to facilitate efficient query evaluations not available with any existing indexing structure to date. These optimisations shall be treated in more depth in chapter 5.

3.3 The PreLevel Encoding Mechanism

The PreLevel encoding mechanism is an extension to the encoding mechanism used in the *XPath Accelerator*. The PreLevel encoding is derived solely on the *preorder traversal* rank and the *level* rank of each node, thus avoiding the need to precompute the *size* information for every node at document loading time as in the encoding mechanism presented in [GST04]. The *level* (or depth) function takes one parameter, a node, and returns the *level* rank value of the node. Figure 3.2 depicts a sample XML document and the corresponding XML document tree with illustrated *preorder* and *level* ranks.



(a) Sample XML document (b) pre/level encoded XML tree.

Figure 3.2: A sample XML document and the associated PreLevel encoded tree as used in XLIM.

Thus, *level(v)* = m if the path from the root of the tree to the node v has length m, for example, *level(a)* = 0 and *level(f)* = 2. The XPath *partition* property introduced in §3.1 is preserved by the combined *preorder traversal* and *level* rank encoding. The remaining XPath axes (**parent**, **child**, **descendant-or-self**, **ancestor-or-self**, **following-sibling** and **preceding-sibling**) determine either supersets or subsets of one of the *primary* axes and may be evaluated from them (this will be demonstrated in §4.4). The following subsections introduce the newly defined conjunctive range predicates based on the combined *preorder traversal* and *level* rank encoding for location steps on the *primary* axes. These conjunctive range predicates form the theoretical foundation upon which the new index structures and algorithms presented in chapters 4 and 5 are built.

Interval Notation

In the *interval* notation adopted in this thesis, open intervals (a,b) are defined as $\{x : a < x < b\}$, they do not contain their endpoints. Closed intervals [a,b] are defined as $\{x : a \leq x \leq b\}$, they contain their endpoints. Open and closed intervals are also open and closed sets respectively.

3.3.1 Navigating the Descendant Axis

The **descendant** axis selects all children of the given context node, and their children recursively, with the resulting nodes in document order [Wor05b]. The new conjunctive

range predicate defining a location step on the **descendant** axis, based on the PreLevel encoding, is as follows:

Lemma 3.1

$$
\begin{aligned}
v \in c/descendant \quad \Leftrightarrow \quad & pre(v) > pre(c) && (i)\\
\wedge \quad & level(v) > level(c) && (ii)\\
\wedge \quad & \forall x : pre(x) \in (pre(c)\,, pre(v)) && (iii)\\
& \Rightarrow level(x) \neq level(c)
\end{aligned}
$$

Lemma 3.1 states that an arbitrary node v is a **descendant** of a given context node c if and only if:

(i) the *preorder* rank of v is greater than the *preorder* rank of c, and

(ii) the *level* rank of v is greater than the *level* rank of c, and

(iii) for all nodes (each individual element of this set is labelled x) having a *preorder* rank greater than *pre(c)* and less than *pre(v)*, that none of those nodes have a *level* rank the same as *level(c)*.

Proof: Condition *(i)* ensures that the *preorder* rank of node v is greater than the *preorder* rank of the context node c. In essence, the first condition exploits the properties of *preorder traversal* to ensure that the arbitrary node v appears, in document order after the given context node c. Condition *(ii)* ensures the *level* rank of node v is greater than the *level* rank of node c. Conditions *(i)* and *(ii)* are intuitive if node v is to be a **descendant** of node c. The third condition ensures that node v does not have another **ancestor** at the same *level* as the given context node c. If there is another **ancestor** at the same *level* as the context node c, then the context node could not be the **ancestor** of node v. This can be stated with certainty due to the properties of *preorder traversal* - namely that a node is visited immediately before its children, and the children are visited from left to right. So, if there is another node at the same *level* as node c, then that node must have a higher *preorder* rank than node c but also a *preorder* rank less than node v (the range requirement of condition *(iii)* ensures this). Thus, although the identity of the **ancestor** at *level(c)* has not been definitely established, it has been definitively determined that the **ancestor** of node v cannot be node c - by finding any other node at the same *level* and

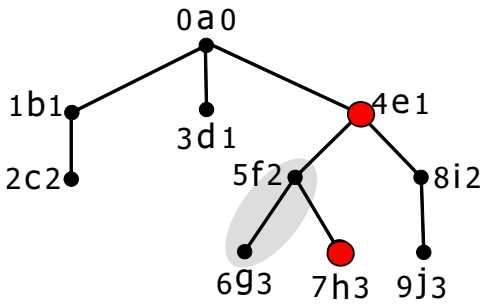


Figure 3.3: Example of navigating the descendant axis of a PreLevel encoded XML tree.

within the range specified. Only if there is no node at the same *level* as the context node c and within the range specified, can it be stated with certainty that the context node c is an **ancestor** of node v, and conversely that node v is a **descendant** of the context node c. Thus, conditions *(i)*, *(ii)* and *(iii)* together ensure that an arbitrary node v is a member of the **descendant** axis of a given context node c.

An illustration of Lemma 3.1 now follows. While referring to the conjunctive range predicate in Lemma 3.1 and to the illustration in Figure 3.3; let v = node h; let c = node e. To determine if node h is a **descendant** of the context node e, one must examine the conditions:

(i) Is *pre(h)* > *pre(e)*...(7 > 4)...condition holds true.

(ii) Is *level(h)* > *level(e)*...(3 > 1)...condition holds true

(iii) For all nodes whose *preorder* rank is greater than *pre(e)* and less than *pre(h)*, these nodes are located within the shaded area in Figure 3.3, does this set contain a node with a *level* rank the same as *level(e)*, in this case 1? No, the set does not contain any nodes with a *level* rank equal to *level(e)* and therefore, the condition holds true.

Conditions *(i)*, *(ii)* and *(iii)* are true. Thus, node h is a **descendant** of the context node e.

Now, take an example whereby the conjunctive range predicate will return false. By following the above example, but assigning node d to be the context node c, conditions

(i) and *(ii)* hold true, but condition *(iii)* fails because node e has the same *level* rank as node *d*.

3.3.2 Navigating the Ancestor Axis

The **ancestor** axis selects all nodes in the document that are ancestors of a given context node. Thus, the conjunctive range predicate defining a location step on the **ancestor axis** based on the PreLevel encoding is:

Lemma 3.2

$$\begin{aligned} v \in c/ancestor \quad &\Leftrightarrow \quad pre(v) < pre(c) && (i) \\ &\wedge \quad level(v) < level(c) && (ii) \\ &\wedge \quad \forall x : pre(x) \in (pre(v)\,,\, pre(c)) && (iii) \\ &\qquad \Rightarrow level(x) \neq level(v) \end{aligned}$$

Lemma 3.2 states that an arbitrary node v is an **ancestor** of a given context node c if and only if:

(i) the *preorder* rank of v is less than the *preorder* rank of c, and

(ii) the *level* rank of v is less than the *level* rank of c, and

(iii) for all nodes (each individual element of this set is labelled x) having a *preorder* rank greater than *pre(v)* and less than *pre(c)*, that none of those nodes have a *level* rank the same as *level(v)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears in document order before the given context node c. Condition *(ii)* exploits the *level* rank properties to ensure node v appears higher in the document tree than node c . Condition *(iii)* ensures that the given context node c does not have another **ancestor** at the same *level* as the node v. If there is any other node at the same *level* as node v, then node v could not be the **ancestor** of the context node c. This can be stated with certainty for the same reason as demonstrated in the proof to Lemma 3.1. The third condition does not identify precisely which other node is the **ancestor**, but simply verifies if some other node other than node v is the **ancestor**. Only if there is no node at the same *level* as

node v and within the range specified, can it be stated with certainty that node v is an **ancestor** of the context node c. Thus, conditions *(i)*, *(ii)* and *(iii)* together ensure that an arbitrary node v is a member of the **ancestor** axis of a given context node c.

An example to illustrate Lemma 3.2 is similar to the example provided for Lemma 3.1. By switching the values assigned to nodes v and c, and referring to Lemma 3.2, walking through the example demonstrates when the conjunction range predicate defining a location step on the **ancestor** axis returns both true and false.

3.3.3 Navigating the Following Axis

The **following** axis selects all nodes that appear after the given context node in document order, excluding the **descendants** of the context node. The new conjunctive range predicate defining a location step on the **following** axis based on the PreLevel encoding is:

Lemma 3.3

$$\begin{aligned} v \in c/following \quad &\Leftrightarrow \quad pre(v) > pre(c) && (i) \\ &\wedge \quad \exists x : pre(x) \in (pre(c)\,,\, pre(v)] && (ii) \\ &\quad\;\; \Rightarrow level(x) \in (0\,,\, level(c)] \end{aligned}$$

Lemma 3.3 states an arbitrary node v is member of the **following** axis of a given context node c if and only if:

(i) The *preorder* rank of v is greater than the *preorder* rank of c, and

(ii) There exists a node whose *preorder* rank is greater than *pre(c)* and less than or equal to *pre(v)* (each individual element of the set is labelled x), such that the *level* rank of x is less than or equal to *level(c)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears in document order after the given context node c. Condition *(ii)* ensures that node v is not a **descendant** of the context node c. The second condition is validated by verifying that there is another node, with a *preorder* rank greater than *pre(c)* and less than or equal to *pre(v)*, and which has a *level* rank less than or equal to the *level* rank of

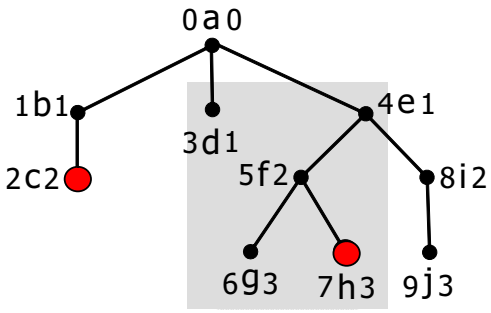


Figure 3.4: Example of navigating the following axis of a PreLevel encoded XML tree.

the context node c. If such a node exists, then due to the properties of *preorder traversal*, the context node c cannot be the **ancestor** of node v, and conversely node v cannot be the **descendant** of the context node c. Thus, conditions *(i)* and *(ii)* together ensure that an arbitrary node v is a member of the **following** axis of given context node c.

An illustration of Lemma 3.3 now follows. While referring to the conjunctive range predicate in Lemma 3.3 and to the illustration in Figure 3.4; let v = node h; let c = node c. To determine if node h is a member of the **following** axis of the context node c, one must examine the conditions:

(i) Is *pre(h)* > *pre(c)*...(7 > 2)...condition holds true.

(ii) For all nodes whose *preorder* rank is greater than *pre(c)* and less than or equal to *pre(h)*, these nodes are located within the shaded area in Figure 3.4, does this set contain a node with a *level* rank less than or equal to *level(c)*? Yes, the set contains several nodes with a *level* rank less than or equal to *level(c)* (nodes d, e and f) and therefore, the condition holds true.

Since both conditions are true, node h is a member of the **following** axis of the given context node c.

Finally, let us take an example whereby the conjunctive range predicate will return false. By following the above example, but assigning node e to be the context node c, condition *(i)* holds true, but condition *(ii)* fails because there is no node within the set that has a *level* rank less than or equal to *level(e)*. The fact that condition *(i)* held true but condition (ii) failed indicates that node e must be an **ancestor** of node h.

3.3.4 Navigating the Preceding Axis

The **preceding** axis selects all nodes in document order that appear before the given context node, excluding all **ancestors** of the context node. The conjunctive range predicate, based on the PreLevel encoding, defines a location step on the **preceding** axis as follows:

Lemma 3.4

$$
\begin{aligned}
v \in c/preceding \quad &\Leftrightarrow \quad pre(v) < pre(c) && (i)\\
&\wedge \quad \exists x : pre(x) \in (pre(v)\,,\, pre(c)] && (ii)\\
&\quad\;\; \Rightarrow level(x) \in (0\,,\, level(v)]
\end{aligned}
$$

Lemma 3.4 states that an arbitrary node v is member of the **preceding** axis of a given context node c if and only if:

(i) The *preorder* rank of v is less than the *preorder* rank of c, and

(ii) There exists a node whose *preorder* rank is greater than *pre(v)* and less than or equal to *pre(c)* (each individual element of the set is labelled x), such that the *level* rank of x is less than or equal to *level(v)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears, in document order, before the given context node c. Condition *(ii)* ensures that node v is not an **ancestor** of the context node c. Due to the properties of *preorder traversal*, the existence of any other node which has a *preorder* rank greater than *pre(v)* and less than or equal to *pre(c)*, and which has a *level* rank less than or equal to *level(v)*, rules out any possibility that node v is the **ancestor** of node c. Thus, conditions *(i)* and *(ii)* ensure that an arbitrary node v is a member of the **preceding** axis of given context node c.

3.4 Conclusions

This chapter began with an in-depth review of the XPath specification and its special *partition* property, together with an overview of tree traversals as they relate to the encoding of XML documents. The new tree encoding mechanism based solely on the combined *preorder traversal* and *level* ranks was then presented. The PreLevel encoding encapsulates

the semantics of the XPath document regions, an important prerequisite to the evaluation of XPath traversals. This was followed by the introduction of newly defined conjunctive range predicates facilitating the evaluation of location steps on the *primary* XPath axes, and the corresponding proofs of their derivation were presented. The conjunctive range predicates and their proofs form the theoretical foundation upon which our new index structure is built. This foundation provides a contribution in establishing the validity and completeness of the PreLevel encoding to fully support the XPath data model and to provide a sound basis on which to construct an XML index structure.

In the next chapter, our new index structure is presented and its core properties are analysed. Furthermore, it will be demonstrated how the new index structure supports the evaluation of location steps on all XPath axes in constant time.

Chapter 4

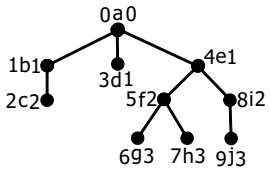
PreLevel Index Structure

In the previous chapter, the PreLevel encoding mechanism was introduced and the theoretical foundations underpinning its support for the XPath specification were presented. In this chapter, a tabular representation is presented based on the PreLevel encoding that facilitates the efficient evaluation of XPath expressions. The tabular encoding is adapted from the XPath Accelerator originally proposed in [Gru02] and enhanced to incorporate the Extended Preorder Index and Level Index now presented. The properties of this tabular encoding are examined in detail. This is followed by the presentation of algorithms facilitating the evaluation of location steps on the *primary* XPath axes in constant time. Furthermore, demonstrations of the evaluation in constant time of the remaining XPath axes (ignoring namespace and attribute) are provided. Finally, a key enabler to the implementation of efficient XPath queries based on the tabular encoding is presented - namely the efficient evaluation of subtree sizes.

The chapter is structured as follows: in §4.1, the new tabular encoding is introduced and its properties explored. In §4.2 and §4.3, the evaluation of location steps on the **descendant** and **following** axes respectively are presented, followed by the demonstration of evaluations on the remaining XPath axes in §4.4. In §4.5, an algorithm to efficiently evaluate subtree sizes is presented and related issues are treated in depth. Finally, in §4.6 we conclude the chapter.

4.1 Indexing Method

The PreLevel tree encoding uses a tabular structure. Subsequently, the PreLevel tabular encoding shall be referred to as the *Extended Preorder Index*. The primary column of the Extended Preorder Index consists of the *preorder* ranks sorted in ascending order. The second column contains the *level* ranks that correspond to the associated *preorder* ranks of the primary column. Extra columns may be added to the Extended Preorder Index to hold further node properties as defined by the XPath/XQuery data model, such as name, node type (node, element, attribute, comment) and more. In particular, to support the **parent** axis in the tabular encoding, a column containing the parent's *preorder* rank of each node, is added to the Extended Preorder Index . However, in order to efficiently evaluate an XPath location step on all the XPath axes, a second index is required. This second index is introduced (hereafter referred to as the *Level Index*) and consists of two columns only, the *level* rank column and the *preorder* rank column. The first column in the Level Index is the *level* rank column, sorted in ascending order, the second column being the *preorder* rank column, again sorted in ascending order. The Extended Preorder Index and Level Index combined may also be referred to as the *PreLevel structure*. A sample PreLevel encoded tree and the corresponding PreLevel structure is illustrated in Figure 4.1.



(a) A pre/level encoded XML tree.

Level	Pre
0	0
1	1, 3, 4
2	2, 5, 8
3	6, 7, 9

(b) Level Index.

Pre	Level	Pos	Parent	Kind	Name	...
0	0	1		Elem	A	
1	1	1	0	Elem	B	
2	2	1	1	Elem	C	
3	1	2	0	Elem	D	
4	1	3	0	Elem	E	
5	2	2	4	Elem	F	
6	3	1	5	Elem	G	
7	3	2	5	Elem	H	
8	2	3	4	Elem	I	
9	3	3	8	Elem	J	

(c) Extended Preorder Index

Figure 4.1: A sample XML tree with corresponding PreLevel structure.

4.1.1 Properties of Extended Preorder and Level Indexes

The PreLevel structure [OBR05] facilitates efficient XPath query processing and evaluations. However, before these contributions may be presented in detail, it is necessary to review the core properties of these indexes.

(i) We acknowledge the high computational costs of processing large XML documents. However, the read-only indexes are constructed once during document parsing and thus, have no runtime construction costs. The support for index updates is addressed as part of the future work in the conclusions to this thesis. Also, the *preorder traversal* ranks and the *level* ranks of a node may be determined during the initial parsing of the XML document tree. Given that all document indexing methods necessarily require the document be first parsed before an index is constructed, the availability of the *preorder* and *level* ranks at parsing time ensure the construction overheads are kept to a minimum. This is a important prerequisite in the provision of acceptable and efficient document loading times for a large XML repository. The original XPath Accelerator proposed in [Gru02] requires both a *preorder* and a *postorder traversal* during index construction. The extra requirement of a *postorder traversal* imposes a significant overhead during document loading as outlined in [VFS04]. The updated XPath Accelerator presented in [GST04], uses a *pre/level/size* encoding and requires an extra cost to compute the subtree size of each node during the construction of the index.

(ii) Each node in the XML tree has a single *preorder* rank and a single *level* rank associated with it. Thus, the Extended Preorder Index contains a one-to-one mapping. However, as many nodes may reside at the same *level*, the Level Index contains a one-to-many mapping - it is a inverted index. Each *level* in the Level Index maps to an array of non-consecutive *preorder* ranks sorted in ascending order. Thus, various structures may be employed in the implementation of this index.

(iii) Both the Extended Preorder Index and the Level Index may be constructed

in parallel during the initial parsing of the XML document tree and these indexes will be sorted in ascending order. The act of parsing of an XML document (reading from top to bottom and left to right) corresponds to a preorder traversal. Thus, the Extended Preorder Index is constructed in a sorted list, sorted on the *preorder* rank in ascending order.

It may not be obvious that the Level Index is also constructed in a sorted list. When the *preorder* traversal begins, the *level* information is recorded also (*level* 0 for the root node). As the *preorder traversal* progresses, all new *levels* and the associated *preorder* ranks are recorded. As the *preorder traversal* encounters nodes on a *level* already recorded, the *preorder* ranks are simply appended to the list of existing *preorder* ranks at that *level*. Thus, depending on the structure used at implementation time, when the *preorder traversal* has been completed, what remains is a column of unique *level* ranks, sorted in ascending order with each *level* rank pointing to a linked list of *preorder* ranks and each linked list also sorted in ascending order.

Thus, the very act of creating the Level Index results in a table with the primary column (*level* ranks) sorted in ascending order, and the second column (*preorder* ranks) also sorted in ascending order.

(iv) Lastly, in order to facilitate a lookup of the Level Index in constant time, a *position* column is included in the Extended Preorder Index. During the construction of the Level Index, before any *preorder* ranks have been inserted, each *level* is assigned a counter initialised to zero. As a *preorder* rank is added (or appended) to the Level Index, the counter at that *level* is incremented by one and its value is written in the *position* column of the Extended Preorder Index, in the row of the related *preorder* rank. Thus, the *position* column is constructed in parallel with the other columns of the PreLevel structure and therefore has minimal construction costs associated with it. The *position* value, when obtained using a lookup of the Extended Preorder Index, facilitates a direct jump to a given *preorder* rank within the Level Index in constant time. The *position* column is the key to enabling the evaluation of location steps on

the *primary* XPath axes in constant time and to the optimised evaluations of *level-based* queries (to be introduced in chapter 5).

The remaining issue to be clarified is the computation of the conjunctive range predicates for each of the XPath *primary* axes in constant time. Before the PreLevel structure can make a new meaningful contribution to the indexing and processing of XML data, it must first demonstrate the ability to provide XPath evaluations that are, at the very least, on par with the existing state-of-the-art preorder encoding schemes, such as the XPath Accelerator reviewed in chapter 2.

4.2 Descendant and Ancestor Axes Evaluation

Due to the hierarchical structure of XML data, the efficient evaluation of **ancestor-descendant** relationships between tree nodes has a significant impact on XPath query performance. In this section, a high level algorithm detailing the steps to evaluate a location step on the **descendant** axis in constant time is provided (in Algorithm 4.1). This algorithm is then illustrated to show how constant time evaluation is possible.

4.2.1 Descendant Lookup Operations

The sample PreLevel encoded tree and the corresponding PreLevel structure are illustrated in Figure 4.1. In order to demonstrate Algorithm 4.1, let v = node h; let c = node e. With a simple lookup of the Extended Preorder Index, it can be verified that *pre(h)* is greater than *pre(e)* (i.e. 7>4), and that *level(h)* is greater than *level(e)* (i.e. 3>1). The Level Index is used to identify the next *preorder* rank greater than *pre(e)* at *level(e)* (i.e. *null*). This information is obtained in constant time as the *position* column of the Extended Preorder Index facilitates a direct jump to *pre(e)* within the *level(e)* index. Note, the next *preorder* rank greater than *pre(e)* at *level(e)*, should it exist, must appear immediately after *pre(e)* because the index is sorted in ascending order. If the next *preorder* rank after *pre(e)* at *level(e)* is greater than *pre(h)*, the node being tested, then node h must be a **descendant** of node e. This can be stated with certainty as the properties of *preorder traversal* require a node's children to be visited immediately after its parent.

```
1 /* The steps below to determine if an arbitrary node v is a
2 descendant of a given context node c is independent of the actual
3 number of descendants and may be evaluated in constant time. */
4 Name:     IsNodeDescendant
5 Given:    An arbitrary node v, a context node c.
6 Returns:  Boolean (TRUE or FALSE)
7 begin
8     //Using the Extended Preorder Index
9     if (pre(v) <= pre(c)) or (level(v) <= level(c)) then
10        return FALSE;
11    endif
12    //Using the Level Index
13    next_pre := next preorder rank after pre(c) at level(c);
14    if (next_pre > pre(v)) or (next_pre == null) then
15        return TRUE;
16    else
17        return FALSE;
18    endif
19 end
```

Algorithm 4.1: To determine if an arbitrary node v is a **descendant** of a given context node c.

Also, if there are no *preorder* ranks greater than *pre(e)* at *level(e)*, indicated with *null*, node h must be a **descendant** of node e. The fact that there may be no *preorder* ranks greater than *pre(e)* at *level(e)* simply means that node e is the root node of the rightmost subtree rooted at *level(e)*.

This subsection has illustrated an evaluation on the **descendant** axis in constant time. However, just as the conjunctive range predicates defining location steps on the **descendant** and **ancestor** axes are similar, an evaluation of a step location on the **ancestor** axis in constant time may be illustrated in a similar fashion by adapting the algorithm appropriately.

4.3 Following Axis Evaluation

The conjunctive range predicates that define a location step on the **following** axis, presented in Lemma 3.3, contain two range-based predicates in condition *(ii)* that must be satisfied. It may not be clear as to how they may be evaluated in constant time. However, by exploiting the properties of the PreLevel structure, constant time evaluation may be demonstrated.

The second condition of the conjunctive range predicate for a location step on the **following** axis requires verification that an arbitrary node *v,* residing within a specific *preorder* interval, has a *level* rank that resides within a specific *level*-based interval. This is quite unlike the conjunctive range predicates for the **ancestor** and **descendant** axes, each of which requires an *equality* test based on the *level* rank. However, according to the XPath specification, the **following** axis selects all nodes that appear after the given context node in document order, excluding the **descendants** of the context node. Thus, using the PreLevel structure, it is sufficient to verify the *preorder* rank of node v is greater than the *preorder* rank of the context node c and that node v is not a **descendant** of the context node c. This can be evaluated in constant time using the PreLevel structure, as demonstrated in §4.3.1.

4.3.1 Constant Time Evaluation

An algorithm demonstrating the steps to evaluate a location step on the **following** axis is provided in Algorithm 4.2. Lines *10-12* verify condition *(i)* of Lemma 3.3 and ensure that node v appears in document order after the context node c. Lines *13-14* determine if node v lies within the *level-based* interval specified in condition *(ii)* of Lemma 3.3. If node v lies within this *level-based* interval, then it can not be a **descendant** of the context node c and therefore must be a member of the **following** axis of the context node c. Lines *15-23* are executed only if node v has a *level* rank greater than the context node c (i.e., is lower down the XML tree) and verify that node v is not a **descendant** of the context node c.

As with the **ancestor** and **descendant** axes, an evaluation of a location step on the **preceding** axis in constant time may be illustrated in a similar fashion to the evaluation of the **following** axis. This is due to the mirror-like relationships between the **ancestor** and **descendant** axes and the **following** and **preceding** axes which are a direct consequence of the XPath *partition* property defined in the XPath specification, as introduced in §3.1.

```
1  /* The steps below to determine if an arbitrary node v is
2  a member of the following axis of a given context node c
3  is independent of the size of the subtree rooted at node
4  c and may be evaluated in constant time. */
5  Name:     IsNodeFollowing
6  Given:    A context node c, an arbitrary node v.
7  Returns:  Boolean (TRUE or FALSE)
8  begin
9     //Using the Extended Preorder Index
10    if (pre(v) <= pre(c)) then
11        return FALSE;
12    endif
13    if (level(v) <= level(c)) then
14        return TRUE;
15    else
16        //Using the Level Index
17        next_pre := next preorder rank after pre(c) at level(c);
18        if (next_pre > pre(v)) or (next_pre == null) then
19            return FALSE;
20        else
21            return TRUE;
22        endif
23    endif
24 end
```

Algorithm 4.2: To determine if an arbitrary node v is a member of the **following** axis of a given context node c.

4.4 Remaining XPath Axes Evaluation

Evaluations of location steps on the four *primary* XPath axes in constant time have been demonstrated in the previous sections. In this section, evaluations in constant time of location steps on the remaining XPath axes (namely **parent**, **child**, **descendant-or-self**, **ancestor-or-self**, **following-sibling** and **preceding-sibling**) are demonstrated.

A location step can only be taken in reference to a context node. Thus, in order to evaluate a location step on any axis, both an arbitrary node and a context node must be given. The context node, according to the XPath specification, constitutes the **self** axis. Therefore, it can easily be shown that the **descendant-or-self** and **ancestor-or-self** axes may be evaluated in constant time by virtue of the evaluations of the **descendant** and **ancestor** axes demonstrated in §4.2. The **parent** axis may be evaluated in constant time using a lookup of the parent column in the Extended Preorder Index. The **child** axis may be evaluated in constant time by verifying the arbitrary node is a **descendant** of the context node and that the *level* rank of the arbitrary node is one greater than the *level* rank of

the context node. The *level* ranks may be identified in constant time using a lookup of the Extended Preorder Index. The **following-sibling** and **preceding-sibling** axes may be evaluated in constant time by verifying that the arbitrary node is a member of the **following** and **preceding** axes respectively (presented in §4.3) and also that the *level* rank of the arbitrary node equals that of the context node.

4.5 Evaluating the size of a Subtree

Although the evaluation of location steps on all XPath axes in constant time using the PreLevel structure can be demonstrated, the real challenge for XPath and XQuery processors and query optimisers is to identify *all* members of an XPath axis as efficiently as possible. For example, to identify all **descendants** of a given context node, or to identify all members of the **following-sibling** axis of a context node. These are the type of queries that form the backbone of many merge-join algorithms which have been identified as the principle bottleneck to efficient query performance over XML data [ZND+01].

These and other such queries and the algorithms to efficiently implement them are presented in chapter 5. However, before proceeding to chapter 5, a key enabler for efficient query evaluation based on the PreLevel structure must be presented - an algorithm to efficiently evaluate the size of a subtree. In this section, the issues involved in the evaluation of a subtree size are explored and an efficient algorithm enabling its evaluation is presented.

When employing the Extended Preorder Index on an XML document tree, the **descendants** of an arbitrary node v in that tree, due to the properties of *preorder traversal*, may be expressed as an interval of lower and upper *preorder* ranks. The lower bound of this interval is the *preorder* rank (plus one) of the arbitrary node v. The upper bound of the interval is the lower bound plus the size of the subtree rooted at node v. Thus, it may be seen that where a *preorder* encoding scheme is employed, the evaluation of a node's **descendants** is equivalent to the evaluation of the size of a subtree rooted at that node.

4.5.1 Computational Cost

Using the PreLevel structure, the size of a subtree tree rooted at an arbitrary node v can be determined efficiently. The evaluation of the subtree size is independent of the actual size of the subtree rooted at node v (and indeed independent of the size of the entire document tree) but rather dependent on the number of levels between the given node v and the root node of the document tree. In [MBV03], a comprehensive study of over 190,000 XML trees was performed revealing that 99% of all documents had less than 8 levels. The vast majority of the remaining 1% of documents had less than 30 levels, with only a tiny fraction having more than 30 levels. The document with the greatest depth had 135 levels, and on further examination, it was determined that the file was computer-generated and contained structural flaws. Thus, it may be seen that the number of levels (or depth) in an XML tree is sufficiently small so as to be deemed to have a minimal computational impact on the evaluation.

4.5.2 The SizeOfSubtree Algorithm

In this subsection, the algorithm for evaluating the size of a subtree rooted at an arbitrary node v is presented. The size of the subtree evaluated with the algorithm is accurate and no extra information beyond the *preorder* and *level* ranks are necessary to determine the size of the subtree. An algorithm *SizeOfSubtree* demonstrating the evaluation of the size of a subtree rooted at an arbitrary node v using the PreLevel structure is provided in Algorithm 4.3.

Lines *14-17* determine if node v is a *leaf* node in constant time using a lookup of the Extended Preorder index. Line *20* identifies the next *preorder* rank after *pre(v)* at *level(v)* in constant time by using the *position* column of the Extended Preorder index to jump directly to *pre(v)* at level(v). For each *level* between the given node v and the root node of the entire document tree, lines *33-41* identify the first node with a *preorder* rank greater than *pre(v)*. The computational processing required at each level is constant. The **parent** column contained in the Extended Preorder Index is used to identify the first node with a *preorder* rank greater than *pre(v)* at any *level*, by identifying the first node with a

Algorithm 4.3: To determine the size of a subtree rooted at an arbitrary node v.

```
1
2  /* The steps below to evaluate the size of a subtree rooted at
3  an arbitrary node v is independent of the actual size of the
4  subtree itself (and indeed the size of the entire document tree),
5  but dependent on the number of levels between the given node v
6  and the root node of the document tree. */
7  Name:     SizeOfSubtree
8  Given:    An arbitrary node v,
9            The maximum preorder rank in document tree: max_pre.
10 Returns:  subtree_size
11 begin
12     //Using the Extended Preorder Index,
13     //determine if node v is a leaf node
14     if (level(pre(v) + 1) <= level (v)) then
15         subtree_size := 1;
16         return subtree_size;
17     endif
18
19     //Using the Level Index
20     next_pre := next preorder rank after pre(v) at level(v);
21
22     //limit will contain the maximum upper preorder rank of
23     //the preorder interval (non-inclusive) specifying
24     //the subtree nodes.
25     limit := next_pre;
26
27     //par(v) returns the preorder rank of the parent node of v
28     par_pre := par(v);
29
30     //For each level between level(v) and the root node,
31     //find the first node with preorder rank > pre(v)
32     init_level := level(v) - 1;
33     for (count = init_level; count > 0; count --)
34         next_pre := next preorder rank after par_pre at level(par_pre);
35         if (limit != null) then
36             if (next_pre != null) and (next_pre < limit) then
37                 limit := next_pre;
38             endif
39         else
40             limit := next_pre;
41         endif
42         par_pre := par(par_pre);
43     endfor
44
45     if (limit != null) then
46         subtree_size := limit - pre(v);
47     else
48         subtree_size := (max_pre - pre(v)) + 1;
49     endif
50     return subtree_size;
51 end
```

preorder rank greater than any of node v ancestors at the required *level*. After processing all specified levels, the size of the subtree may be evaluated as the difference between the first *preorder* rank greater than *pre(v)* and the *preorder* rank of node v itself.

Thus, the processing requirements of the algorithm is independent of:

1. the number of nodes at each *level*.
2. the size of the subtree rooted at node v.
3. the size of the entire document tree.

The only variable is the number of levels to be processed by the algorithm. However, the comprehensive study in [MBV03] referenced earlier in this section revealed the substantial majority of XML document trees have less than 30 levels, a number still significantly small as to impose minimal computational overheads, except for unusual circumstances.

4.5.3 Observations

If there is no node greater than *pre(v)* at any of the levels specified (between the *root* node and node v), then node v and its ancestors simply reside at the right-most branch of the entire document tree at those levels. The term *null* is used to indicate that no greater *preorder* rank was found. In the Algorithm *SizeOfSubtree*, the given node itself is included in the size of the subtree and thus, the smallest value returned is 1. The algorithm can be modified not to include the given node in the size of the subtree, should an implementation require a return value of zero for leaf nodes.

4.6 Conclusions

In this chapter, a tabular representation for the PreLevel encoding was presented - namely the PreLevel structure. The PreLevel structure was shown to have minimal computational overhead associated with its construction as both *preorder* and *level* ranks are available during document parsing. Its indexes may be constructed in parallel and are automatically sorted in ascending order during the construction phase. Algorithms for evaluating

location steps on the **descendant** and **following** axes in constant time were presented and illustrated, and from these, evaluations on the remaining XPath axes in constant time were demonstrated. The ability of the PreLevel structure to support the evaluation of location steps on all XPath axes in constant time is an important contribution in that it demonstrates, thus far, that the PreLevel structure is *on par* with the best existing indexing structures to date, such as the XPath Accelerator.

Furthermore, to unlock the real potential of efficient query evaluations based on the PreLevel structure, an algorithm to efficiently evaluate the size of a subtree was presented. In chapters 3 and 4, the theoretical and (index) structural foundations of the PreLevel structure have been fully laid down. The next step is to demonstrate the various classes of efficient queries possible, based on the PreLevel structure.

Chapter 5

XPath Query Optimisations

In chapter 4, the PreLevel structure was introduced and it was demonstrated how this structure supported the evaluation of location steps on all XPath axes in constant time. The purpose of this chapter is to detail the principle benefits of the PreLevel structure. This is achieved by demonstrating the efficient evaluation of *all* members of the primary XPath axes and then introducing the key contribution of the PreLevel structure: level-based optimised queries. An overview of level-based queries is provided, followed by a detailed explanation of their small computational costs. Several classes of XPath queries that can be accommodated by level-based queries are introduced and their efficient evaluations demonstrated. Lastly, three benefits made available to XPath and XQuery processors through level-based optimised queries are described.

The chapter is structured as follows: In §5.1, the efficient evaluation of *all* members of the *primary* XPath axes are demonstrated, together with their computational costs. In §5.2, the optimised level-based queries are introduced, their computational costs discussed, the various classes of optimised queries supported are detailed and the added benefits to XPath and XQuery processors outlined. Finally, we offer our conclusions in §5.3.

5.1 Primary XPath Axes Evaluation

In chapter 4 it was shown that the PreLevel structure supports the evaluation of location steps on all XPath Axes. However, XPath query processors often have to evaluate *all*

members of an XPath axis. This is especially true for XQuery processors which must evaluate XPath expressions embedded in XQuery statements. The PreLevel structure facilitates the efficient evaluation of all members commencing from an arbitrary context node, without bias as to where the context node occurs in the XML document tree.

XPath Accelerator Comparison The XPath Accelerator facilitates the evaluation of *all* members of the primary XPath axes in constant time. This is possible because the XPath Accelerator precomputes the size of the subtree rooted at every node during index construction. The size of a subtree is the key determinant in the evaluation of all members of the primary XPath axes.

5.1.1 Descendant and Ancestor Axes Evaluations

Algorithm 5.1 illustrates the steps to evaluate *all* members of the **descendant** axis of an arbitrary node v. The steps required to determine all **descendants** of node v is independent of the size of the subtree rooted at node v, and is also independent of the size of the entire document tree, but is dependent on the number of levels between the arbitrary node v and the root node of the document tree.

Lines *14-17* determine if the arbitrary node v is a leaf node (i.e. has no children) and return the *empty* sequence if true. Leaf node status can be determined in constant time using a lookup of the Extended Preorder Index. Line *20* uses the *SizeOfSubtree* algorithm presented in §4.5 to determine the maximum *preorder* rank of the **descendants** of node v. Lines *23-24* identify (in constant time) the lower and upper *preorder* ranks of the interval containing the **descendants** of node v. All steps in the algorithm *AllDescendants* may be computed in constant time, with the exception of the *SizeOfSubtree* function. However, the computational cost of the *SizeOfSubtree* algorithm has already been shown to be minimal because the processing required at each level is constant. Thus, the PreLevel structure facilitates an efficient evaluation of all members of the **descendant** axis of an arbitrary node v.

In a similar manner, the computational costs to evaluate all members of the **ancestor** axis of an arbitrary node v is solely dependent on the number of levels between the given

```
1  Name:     AllDescendants
2  Given:    An arbitrary node v,
3            The maximum preorder rank in document tree: max_pre.
4  Returns:  A sequence of document nodes labelled descendants
5            or the empty sequence.
6  begin
7     //Using the Extended Preorder Index
8     if (level(pre(v)+1) <= level(v)) then
9         //There are no descendants
10        return empty;
11    endif
12
13    //Using algorithm to determine size of subtree
14    subtree_size := SizeOfSubtree(v);
15
16    //Now identify the interval containing the descendants
17    last_pre := pre(v) + (subtree_size - 1);
18    descendants := all nodes in interval (pre(v),last_pre];
19    return descendants;
20 end
```

Algorithm 5.1: To determine all **descendants** of an arbitrary node v.

node v and the root node of the entire document tree. In order to identify all **ancestors** of an arbitrary node v, a single lookup of the parent column of the Extended Preorder Index for each level between *level(v)* and the root node is required.

5.1.2 Following Axis Evaluation

The **following** axis selects all nodes that appear after the given context node in document order, excluding the **descendants** of the context node. Algorithm 5.2 illustrates the steps to evaluate *all* members of the **following** axis of an arbitrary node v.

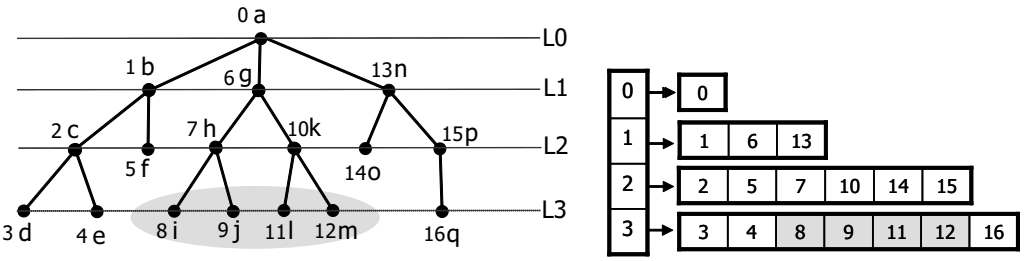
Lines *18-19* are executed if node v is a leaf node and identifies all members of the **following** axis as all nodes having a *preorder* rank greater than *pre(v)*. Line *23* uses the *SizeOfSubtree* function to determine the largest *preorder* rank of the **descendants** of node v. Line *24* identifies all members of the **following** axis as all nodes having a *preorder* rank greater than the largest *preorder* rank of node v's **descendants**. All steps in the *AllFollowing* algorithm may be computed in constant with the exception of the *SizeOfSubtree* function. Thus, the PreLevel structure facilitates an efficient evaluation of all members of the **following** axis of an arbitrary node v.

```
1  Name:      AllFollowing
2  Given:     An arbitrary node v,
3             The maximum preorder rank in document tree: max_pre.
4  Returns:   A sequence of document nodes labelled followingNodes
5             or the empty sequence.
6  begin
7     //Using algorithm to determine size of subtree
8     subtree_size := SizeOfSubtree(v);
9
10   //If leaf node
11    if (subtree_size == 1) then
12        followingNodes := all nodes in interval (pre(v),max_pre];
13        return followingNodes;
14    endif
15
16    // if not leaf node
17    Let maxSubPre := pre(v) + (subtree_size - 1);
18    followingNodes := all nodes in interval (maxSubPre,max_pre];
19    return followingNodes;
20 end
```

Algorithm 5.2: To determine all members of the **following** axis of an arbitrary node v.

5.1.3 Preceding Axis Evaluation

The **preceding** axis selects all nodes in document order that appear before the context node, excluding all **ancestors** of the context node. Thus, the set of all members of the **preceding** axis can be identified as the *set of all nodes before the context node* minus the *set of ancestors*. In formal set theory notation, the set of all members of the **preceding** axis is defined as the *relative complement* of the *set of ancestors* in the *set of all nodes before the context node*. The *set of all nodes before the context node* can be determined in constant time to be all nodes with a preorder rank less than the preorder rank of the context node. The size of the *set of ancestors* is equal to the number of levels between the root node and the context node. The comprehensive study performed in [MBV03] (as discussed in §4.5.1) revealed that the number of levels in the vast majority of XML documents will be below 30, a relatively small value. Thus, as the *set of all nodes before the context node* will be a *preorder* defined interval, sorted in ascending order, the subtraction of a (relatively) small number of **ancestor** *preorder* ranks from this set will add minimal computation overhead to the overall evaluation. Thus, the PreLevel structure facilitates an efficient evaluation of all members of the **preceding** axis of an arbitrary node v.



(a) PreLevel encoded XML tree with grandchildren of node g selected.

(b) Level index with grandchildren of node g highlighted.

Figure 5.1: An illustrated level-based query to select all grandchildren of node g.

5.2 Optimised Level-based Queries

The intrinsic properties of the PreLevel structure may be exploited to provide optimised level-based queries. A *level-based* query is such that the results of the query reside at a particular level in the XML tree. In this section, an overview of level-based queries follow and their computational costs are discussed and finally, the classes of queries they support are highlighted.

5.2.1 Level-based Queries Overview

Taking the **descendant** axis as an example, all nodes that are a **descendant** of an arbitrary node v will reside in a *preorder-defined* interval, delimited by lower and upper *preorder* ranks. Thus using the Level Index, it is easy to identify a sequence of nodes residing at a particular *level* that belong to a *preorder-defined* interval. For example, given a query to select all grandchildren of an arbitrary node v; the result of such a query will be represented using the Level Index as an interval of lower and upper *preorder* bounds residing at a specific *level*. Figure 5.1(a) depicts a sample PreLevel encoded XML tree with the grandchildren of node g selected. Figure 5.1(b) illustrates the corresponding Level Index with the highlighted *preorder-defined* interval containing node g's grandchildren. The *position* column of the Extended Preorder Index facilitates a direct jump to the lower and upper *preorder* bounds within the Level Index.

5.2.2 Computational Cost

The Level Index is sorted in ascending order and can be searched efficiently using a binary search algorithm with a time complexity of $O(log\ n)$ [Sed88]. The lower bound of the *preorder* interval containing node v's **descendants** at a given level l, is obtained by performing a binary search at level l for the first *preorder* rank greater than *pre(v)*. In a similar fashion, the upper bound of the *preorder* interval containing node v's **descendants** at level l, is obtained by performing a binary search at level l for the last *preorder* rank preceding a *container preorder* rank of node v's **descendants**. A *container preorder* rank is a *preorder* rank greater than the largest *preorder* rank in node v's **descendants**. Due to the properties of *preorder traversal*, a valid *container preorder* rank for node v's **descendants** is the next *preorder* rank greater than *pre(v)* at *level(v)*. The *container rank* can be obtained in constant time using a lookup of the Level Index and provides an upper bound for node v's **descendants** at level l.

Given the *preorder* rank of a context node, the lower and upper bounds of the interval encapsulating the set of all members of the context node's **descendants** at an arbitrary level l can be obtained using the Level index, requiring a total of two single binary search operations of time complexity $O(log\ n)$ each, at level l. *The optimisation provided by level-based queries is such that the processing of nodes at intermediary levels is unnecessary for all levels between the context node and the level to be queried (exclusive).* For clarity, a single binary search operation of time complexity $O(log\ n)$ shall be referred to as a *search operation.*

The optimal time complexity for reading n values from an array of size n is linear, i.e. $O(n)$. The results of a level-based query is an array subset of the Level Index, which is always sorted in document order. Thus, given that the *position* column facilitates a direct jump to the lower and upper *preorder* bounds within the Level Index; when both lower and upper bounds of the interval have been obtained, the actual results of the level-based queries may be retrieved in *optimal* time. Once the interval is known, the solution is optimal for retrieving all **descendants** of a given node v that reside at an arbitrary level l.

Algorithm 5.3 illustrates the steps to evaluate all **descendants** of a given node v residing

```
1  /* The steps below to determine all descendants of an arbitrary
2  node v at a given level L requires only two binary searches of
3  time complexity O(log n) each, at level L. */
4  Name:     AllDescendantsAtLevelL
5  Given:    An arbitrary node v,
6            The maximum preorder rank in document tree: max_pre,
7            A level L, where L is the path length from node v
8            to the level queried.
9            (e.g. to find all grandchildren of node v, let L = 2)
10 Returns:  A sequence of document nodes labelled descendants
11           or the empty sequence.
12 begin
13    //Using the Extended Preorder Index
14    if (level(pre(v)+1) <= level(v)) then
15        //There are no descendants
16        return empty;
17    endif
18
19    //Using the Level Index
20    next_pre := next preorder rank after pre(v) at level(v);
21
22    //Convert relative level rank to absolute level rank of doc
23    queryLevel := level(v) + L;
24
25    //Identify the interval containing descendants at queryLevel
26    if (next_pre == null) then
27        descendants := all nodes in interval (pre(v),max_pre] at
28        queryLevel;
29    else
30        start_pre := next preorder value > pre(v) at queryLevel;
31        descendants := all nodes in interval [start_pre,next_pre)
32        at queryLevel;
33    endif
34    return descendants;
35 end
```

Algorithm 5.3: To determine all **descendants** of an arbitrary node v at a given level L.

at an arbitrary level l. Regardless of the value of *next_pre* in line *26*, both sections of code belonging to the *if* statement require only two *search operations* on the Level Index. All other lines in the algorithm may be processed in constant time.

XPath Accelerator Comparison Level-based optimisations are not provided by the XPath Accelerator. These optimisations are made possible by the introduction of an inverted index, the Level Index in our PreLevel structure. While referring to Figure 5.1(a) and (b), an illustration now follows describing how the XPath Accelerator evaluates the query: *select all grandchildren of node g.* The XPath Accelerator uses a *pre/size/level*

encoding for each node in the document tree and thus, has precomputed the size of the subtree rooted at node *g*. Given the size of the subtree, the lower and upper bounds of the *preorder-defined* interval encapsulating all members of the descendants of node *g* may be determined in constant time. The lower bound is *pre(v) plus one* and the upper bound is *pre(v) plus subtree_size*. However, for each descendant of node *g*, the XPath Accelerator must lookup its index to determine the level rank of the descendant. Thus, the evaluation of a level-based query using the XPath Accelerator has a linear time complexity of $O(n)$ where n is the number of descendants of the context node.

5.2.3 Evaluating All Members of the Child Axis

The evaluation of *all* members of the **child** axis of an arbitrary node v, can be easily accommodated by our level-based queries. Given that the **descendants** of an arbitrary node v at a given level l reside in a *preorder* defined interval at level l, it follows that all members of the **child** axis of node v reside in a *preorder* defined interval at *level(v)* plus one in the Level Index. However, due to the properties of *preorder traversal*, the lower bound of the *preorder* interval containing all members of the **child** axis is simply *pre(v)* plus one (constant time evaluation). The identification of the upper bound of the interval requires a single *search operation*. Once the lower and upper bounds are known, the retrieval of all children is optimal (for reasons outlined in §5.2.2). Thus, the PreLevel structure enables the efficient evaluation and retrieval of *all* members of the **child** axis of an arbitrary node v.

For example, while referring to Figure 5.1 and given the context node *b,* the results of the XPath expression *child::** reside in the interval [2,5] at level 2.

XPath Accelerator Comparison The XPath Accelerator determines the lower bound in the same way as the PreLevel structure. The upper bound of the children of node v is evaluated by determining the largest *preorder* rank of its descendants (i.e. *pre(v)* plus subtree size) and finding its ancestor at *level(v)* plus one. Subsequently, all children and their descendants between these two bounds require a lookup of the XPath Accelerator index to determine its level (or its parent). Either the *level* or *parent* value is needed

to determine if a descendant is a child of node *v*. In either case, the processing required is *just under* linear time complexity. The actual optimisation provided by the XPath Accelerator is the pruning of the subtree of the rightmost child of node *v*. All children (except the rightmost child) of node *v* and their descendants must be processed. Thus, the optimisation provided is minimal.

5.2.4 Following-Sibling and Preceding-Sibling Axes Evaluation

The evaluation of *all* members of the **following-sibling** and **preceding-sibling** axes of an arbitrary node *v* can also be accommodated by our level-based queries. The lower bound of the *preorder* interval encapsulating the set of all members of the **following-sibling** axis is the next *preorder* rank greater than *pre(v)* at *level(v)*. The lower bound can be obtained in constant time using a lookup of the Level Index.

The upper bound of the *preorder* interval containing the **following-sibling** axis of an arbitrary node *v* is the largest *preorder* rank at *level(v)*. The Level Index maintains a record of the total number of elements stored at each level (hereafter referred to as the *total* number). The *total* number at each *level* will correspond to the position value of the last element at each *level*, that is, the position value of the largest *preorder* rank at each *level* (because the Level Index is sorted in ascending order). Therefore, the *total* number at each *level* is equal to the *position* value of the upper bound *preorder* rank for each *level*.

Thus, the lower and upper bounds of the *preorder* interval encapsulating the set of all members of the **following-sibling** axis of an arbitrary node *v* are obtained in constant time. Furthermore, given that the results of level-based queries may be retrieved in optimal time, as discussed in §5.2.2, the PreLevel structure supports the evaluation and retrieval of *all* members of the **following-sibling** axis of an arbitrary node *v* in *optimal* time.

In a similar manner, it may be demonstrated that the evaluation and retrieval of all members of the **preceding-sibling** axis of an arbitrary node *v* can be performed in *optimal* time.

For example, while referring to Figure 5.1 and given the context node *h*, the results of the XPath expression *following-sibling::** reside in the interval [10,15] at level 2.

XPath Accelerator Comparison The optimisation provided by the XPath Accelerator in evaluating *all* members of the **following-sibling** axis is minimal and is similar to that outlined in §5.2.3.

The XPath Accelerator provides no optimisation for the evaluation of the **preceding-sibling** axis. The processing requirements has a linear time complexity of $O(n)$ where n is the number of nodes preceding the context node.

5.2.5 Evaluating the Size of a Level-based Result Set

The intrinsic properties of the PreLevel structure allow for the accurate and efficient evaluation of the size of the result set for all level-based queries, without requiring their materialisation.

When the lower and upper *preorder* bounds defining an interval result set at any given *level* are known, the *position* values of the lower and upper *preorder* bounds may be obtained in constant time using a lookup of the Extended Preorder Index. By subtracting the *position* value of the lower bound from the *position* value of the upper bound, we are left with the exact number of elements in the result set. Thus, for all level-based queries, the computational cost of determining the size of the result set is the same as the computational cost of determining the lower and upper bounds of the interval defining the result set, namely: two *search operations*.

The PreLevel structure enables the efficient evaluation of the exact size of the result set of a level-based query without having to materialise the result set. This feature is of particular benefit to the query planners and query optimisers of XQuery processors in their evaluation of query plans for the execution of XPath expressions embedded in XQuery statements.

XPath Accelerator Comparison The XPath Accelerator cannot facilitate level-based optimisations for reasons described in §5.2.2 and thus, the evaluation of the size of a level-based result set has a linear time complexity $O(n)$ where n is the number of descendants of the context node.

5.2.6 Optimising Wildcard Evaluation

The size of an XPath query (in terms of the number of location steps) is a principle determinant of its evaluation complexity, as demonstrated in [GKP02]. Several optimisations have been proposed to minimise the size of an XPath query by eliminating redundant steps [AYCLS01] [Ram02]. In particular, recent research has focused on the optimisation of XPath queries by reducing wildcard steps [CFZ04]. The level-based optimisations provided by the PreLevel structure facilitate the reduction of wildcard steps.

Overview

A wildcard step refers to an XPath location step with the wildcard *nodetest*; examples include **child**::* and **ancestor**::*.. Wildcard steps are employed when the element names are unknown or do not matter. Wildcard steps are also often used as shorthand notation to represent a set of element names. For example, if a publication has a journal or a conference subelement, the query */publication/journal/title* union */publication/conference/title* can be expressed more succinctly using the wildcard based path expression: */publication/*/title* .

Wildcard reduction

In Figure 5.2, an employee schema for a multi-national company is presented. Every element in the schema is mandatory. The following query is used to illustrate wildcard step reduction: *select the names of all employees in England whose salary is greater than one hundred thousand pounds*. Given the context node *England*, this query may be expressed in XPath using wildcard steps as:

- self::node()/*/*/salary[. > 100000]/parent::node()/child::name

This query may be expressed more succinctly using the abbreviated XPath syntax as:

- ./*/*/salary[. > 100000]../name

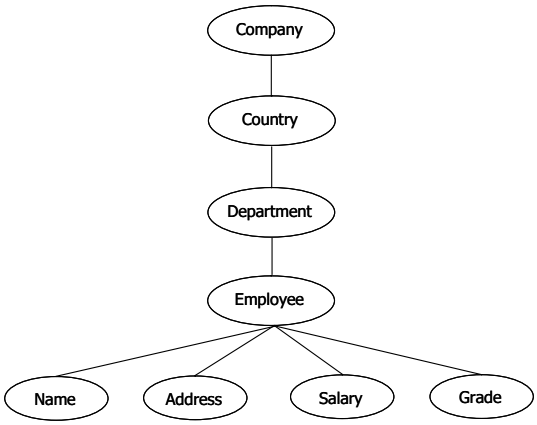


Figure 5.2: Employee Schema

An XPath processor must perform three steps to successfully evaluate this expression.

1. First locate all *salary* elements that are great-grandchildren of the *England* element.
2. From all *salary* elements found, select only those whose value is greater than 100,000 pounds.
3. Identify the *name* of the employees associated with the selected *salary* elements.

In order to demonstrate the wildcard step reductions possible using level-based optimisations, we will focus on the evaluation of the first step. A conventional XPath processor would first have to identify all children of *England* (departments), and then for every *department*, identify all employees working in the departments, and then for every *employee*, identify its *salary*. Using our PreLevel structure and exploiting the level-based optimisations; upon identifying the *preorder* rank of the *England* element, we can quickly determine the lower and upper *preorder* bounds of all *salary* elements that are a **descendant** of *England* at the level *level(England) plus three*. The PreLevel structure does not need to process the *department* (elements) and the employees in order to locate the *salary* elements. The PreLevel structure permits the rewrite of the query to search for *salary* elements at the required level (using the Level Index), thus eliminating the wildcard steps and negating the need for processing at intermediary levels. Furthermore, the lower and upper bounds of the *preorder-defined* interval encapsulating the set of all *salary* elements of all employees in England, may be determined with just two *search operations*.

The example provided illustrated wildcard step reduction using schema-based documents. However, level-based optimised queries may be used to reduce wildcard steps over any type of XML document. Furthermore, wildcard reduction may be used in conjunction with the evaluation of the result set size to enhance an XPath processor during the query planning and query optimisation stages in selecting the best query evaluation strategy.

XPath Accelerator Comparison In order to evaluate the first step of the query, the XPath Accelerator would follow the path outlined for a conventional XPath processor of recursively following the paths of all the descendants of the *England* element until it arrived at the level sought and then begin searching for salary elements. The evaluation of the first step has a linear time complexity $O(n)$ where n is the number of descendants of the *England* element between *level(England)* and *level(England) plus three*.

5.2.7 Efficient Ancestor Evaluation

The evaluation of a node's **ancestor** at an arbitrary *level* may be efficiently performed by recursively identifying the node's **parents** using a lookup of the Extended Preorder Index. Under unusual circumstances, the number of levels in an XML document may be very large, and a node's **ancestor** at any arbitrary level may be obtained using the Level Index and requires a single *search operation*.

Due to the properties of *preorder traversal*, given an arbitrary node v, the **ancestor** of node v at an arbitrary level l, must be the largest *preorder* rank, less than *pre(v)* at level l. The *preorder* rank of the **ancestor** may be obtained by performing a *search operation* at level l for the largest node less than *pre(v)*.

XPath Accelerator Comparison The XPath Accelerator evaluates a node's **ancestor** at an arbitrary level by recursively identifying the node's **parents** using a lookup of its index. Thus, no optimisation is provided.

5.3 Conclusions

In this chapter, the contributions of the PreLevel structure to the efficient evaluation of XPath expressions for XPath and XQuery processors have been presented. We began with a demonstration of the efficient evaluation of *all* members of the *primary* XPath axes, without bias towards the location of the context node within the XML document. The evaluation of all members of the *primary* axes is dependent on the number of levels between the context node and the root node (due to the *SizeOfSubtree* function) but is unaffected by the number of nodes in the document tree. Thus, it follows that the runtime computational costs of the PreLevel structure scale well for very large documents as the number of nodes in the document tree have little impact on the processing requirements of query evaluations.

We then introduced level-based queries and determined that the worst case computational costs of level-based queries required just two binary search operations of time complexity $O(log\ n)$ each. The optimisation provided in the evaluation of level-based queries is such that the processing of all nodes on all levels between the root node and the context node (exclusive) is not required. The elimination of the large scale processing of nodes at intermediary levels is a significant contribution of the PreLevel structure, in comparison to the XPath Accelerator. Furthermore, once the interval containing the results of a level-based query has been identified, the retrieval of the results (respecting document and node order) may be performed in *optimal* time. The evaluation of all members of the **child** axis can be accommodated by level-based queries and requires only one binary search operation of time complexity $O(log\ n)$ for its evaluation. Moreover, the evaluation *and* retrieval of all members of the **following-sibling** and **preceding-sibling** axes may be performed in *optimal* time.

Level-based query optimisations may be exploited to provide many other benefits to XPath and XQuery processors. Three of these benefits are outlined in this chapter: efficient evaluation of the size of the result set, optimised wildcard query evaluation, and the efficient evaluation of a node's ancestor at an arbitrary level.

The PreLevel structure, through the benefits provided by the efficient evaluation of all

members of the *primary* XPath axes in conjunction with optimised level-based queries, provides an efficient structure for the indexing of large XML collections. Indeed, of all classes of queries treated in this thesis (covering all XPath axes evaluations), none of their computational costs are dependent on the number of nodes in the tree. Thus, all large scale XML collections may be efficiently indexed and queried using our PreLevel structure.

In this chapter we have finalised our contribution to the indexing of XML data and have presented an index structure that satisfies our requirements identified in chapter 2. In the next chapter, a summary of the work presented in this thesis is provided and some directions for future work are outlined.

Chapter 6

Conclusions

The aim of this research thesis was to design an indexing structure for large XML repositories that would serve as the core component in the provision of an efficient, scalable and reliable query service. Unlike other research projects, this work focused on providing an index structure that fully supports the XPath and XQuery data models while also providing support for efficient structural and navigational queries in addition to the traditional data-centric and content-based queries. In particular, several classes of queries can benefit from the novel level-based optimisations of our new index structure, optimisations not available with any existing indexing structures to date. A second objective of our research was to ensure our index structure did not specify an implementation model or special storage requirements. In this chapter a review of the thesis is presented in §6.1 and areas of future research are offered in §6.2.

6.1 Thesis Summary

In chapter 1, an introduction to the XML data model was presented and the tree structure underlying semi-structured data was described. The expressive and extensible nature of XML resulted in its rapid adoption and consequently the need for flexible and robust XML repositories and DBMS emerged. In response, the development of XML DBMS has evolved in two broad streams: XML Enabled databases and Native XML databases. The efficiency of the query services offered by these DBMS reflect their underlying data model.

XML Enabled databases perform data-centric queries efficiently but perform poorly in the processing of structural queries as the (object-)relational data model is not a hierarchical model. In a similar manner, Native XML databases have an XML document as their fundamental data unit and process structural queries efficiently while giving poor performance for data-centric queries. The motivation for this thesis emerged from the absence of an XML DBMS facilitating both efficient hierarchical and content-based queries. In particular, the semantically rich nature of XML data whereby the structure, content and description of the data is embedded in the XML file, highlighted the importance of the indexing service in the provision of an efficient query service. Consequently, this research focused on the provision of a new index structure for XML data to support an efficient, scalable and reliable query service in a large XML repository.

In chapter 2, several research projects covering existing *state-of-the-art* indexing structures and techniques were analysed and discussed. When examining these projects, the emphasis was on their support for the XPath and XQuery specifications in addition to the efficient evaluation of structural and navigational queries. The ORDPATH project focused on the provision of a dynamic labelling scheme to the expense of an efficient query service. The indexing mechanism used in the eXist database project drew from the strengths of several research initiatives and provided full support for structural query evaluation from indexes alone. However, the path join algorithms used in eXist operated on a level-order index numbering scheme and resulted in inefficient evaluations when compared to preorder-based index numbering schemes. Lastly, the XPath Accelerator index, which most closely adheres to our indexing service requirements was presented. It has been specifically designed to support the XPath and XQuery specifications. The original XPath Accelerator *pre/post* encoding had several shortcomings, many of which were overcome by the more recent *pre/size/level* encoding adopted by its authors. Nevertheless, the XPath Accelerator still lacks an efficient evaluation mechanism for several classes of queries and thus, motivating the goal of our PreLevel structure to provide several optimisation benefits not currently available with existing indexing structures.

The PreLevel encoding underlying the PreLevel structure was presented in chapter 3. The PreLevel encoding encapsulates the semantics of the XPath document regions, an impor-

tant requisite for the evaluation of XPath traversals. For each of the primary XPath axes, new conjunctive range predicates defining location steps on the axis were presented and the corresponding proofs of their derivation provided. The conjunctive range predicates have been derived from the intrinsic properties of the preorder traversal ranks and level ranks alone.

It should be noted at this point that although the *pre/size/level* encoding of the XPath Accelerator appears similar to the *pre/level* encoding of the PreLevel structure, the underlying mechanism used to evaluate location steps on XPath axes is fundamentally different. The XPath Accelerator exploits the property describing the relationship between the preorder, postorder, size and level attributes of nodes in a tree, as presented in §2.3.3. The PreLevel encoding, on the other hand, evaluates location steps on XPath axes using information derived solely from the properties of the preorder and level attributes of the nodes. Thus, although the XPath Accelerator and the PreLevel structure employ similar node properties, the computational process underlying the evaluation mechanisms of each of the index structures are fundamentally different. The conjunctive range predicates of the PreLevel encoding and their proofs form the theoretical foundation upon which our index, the PreLevel structure, is built. This platform provides a contribution in establishing the validity and completeness of our PreLevel encoding to fully support the XPath data model and to provide the sound basis on which to construct an XML indexing mechanism.

The PreLevel Index structure was presented in chapter 4. The tabular encoding of the PreLevel structure is an extension to the XPath Accelerator and enhanced to incorporate our Extended Preorder and Level indexes. The indexes are constructed in parallel and are automatically sorted during the construction phase. The construction phase was shown to have minimal computational overhead as both the preorder and level ranks are available during document parsing. Algorithms for evaluating location steps on the **descendant** and **following** axes in constant time were presented and illustrated, and from these, evaluations on the remaining XPath axes in constant time were demonstrated. The ability of the PreLevel structure to support the evaluation of location steps on *all* XPath axes in constant time is an important contribution in that it demonstrates, thus far, that the PreLevel structure is *on par* with the best existing indexing structures to date, such as

the XPath Accelerator. Furthermore, a efficient algorithm to evaluate the size of a subtree was presented. This algorithm, whose computational costs are a function of the number of levels in the tree, contributes to the efficient query evaluations presented in chapter 5.

The principle benefits of the PreLevel structure were presented in chapter 5. An efficient evaluation of *all* members of the primary XPath axes commencing from an arbitrary node was demonstrated. The key contribution of the PreLevel structure was then introduced: level-based optimised queries. A level-based query is such that the results of the query reside at a particular level, for example, all grandchildren of a context node. The worst case computational costs for the evaluation for a level-based query is two binary search operations of time complexity $O(log\ n)$ each. The level-based optimisation eliminates the need for large scale processing of nodes at intermediate levels, regardless of how many levels separate the context node and the level required. No XML index structure to date offers such optimisation. This optimisation is made possible by the introduction of an inverted index, the Level index in our PreLevel structure. The **child** axis can be accommodated by our level-based optimisation and requires a single search operation of time complexity $O(log\ n)$. This is more efficient than the evaluation provided by the XPath Accelerator, as outlined in chapter 5. It was further demonstrated that all members of the **following-sibling** and **preceding-sibling** axes may be evaluated and retrieved in *optimal* time, an optimisation not possible with the XPath Accelerator. The size of a level-based result set without requiring its materialisation may be evaluated with just two search operations. This feature is of particular benefit to the query planners and query optimisers of XQuery processors in the evaluation of query plans for the execution of XPath expressions embedded in XQuery statements. Another feature of the PreLevel structure's optimisation is the support for wildcard step reduction and query rewriting and examples were provided in chapter 5. The PreLevel structure, through the benefits provided by the efficient evaluation of all members of the XPath axes in conjunction with the optimised level-based queries, provides a powerful indexing structure for a scalable, reliable and effective query service for large scale XML repositories.

6.2 Future Research

As native XML databases become more pervasive and industrial object-relational databases incorporate XML support as a core component in their product range, more and more organisations are employing XML within their information management and exchange strategies. However, the SQL data format still dominates the mainstream information database technology domain and it is envisaged that it will continue to do so for some time to come. The seamless co-existence of SQL and XML data side by side is becoming ever more likely. In tandem with the heterogeneity of data formats deployed by organisations, many applications have requirements that their underlying DBMS support updates. A limitation of our PreLevel structure presented in this thesis is its read-only nature. The ability to support data insertions and deletions is of critical concern to many application domains. The final segment of this thesis focuses on future directions for continuing this research.

6.2.1 RDBMS Compatibility

Due to the tabular representation of the PreLevel structure, it can reside inside a relational database - it is effectively a relational index structure. Its implementation can benefit from the well established indexing technology of the relational domain, notably the B+tree and the R-tree. It should be noted at this point that the PreLevel structure was presented as a theoretical model and the benefits of the PreLevel structure such as constant time lookup were demonstrated with respect to this theoretical model. However, depending on the techniques employed at implementation time, a hash table index will permit access in constant time, whereas a B+tree index enables access time that is a logarithmic function of the document size. Thus, the runtime performance benefits provided by the PreLevel structure is determined in part by the implementation techniques employed at deployment.

When the PreLevel structure was introduced, it was presented as an extension to the XPath Accelerator. The Extended Preorder index of the PreLevel structure has a similar tabular encoding to the XPath Accelerator. In [GVT04], the author provides an in-depth analysis of the relational aspects of the XPath Accelerator and outlines several experiments using

various RDBMSs demonstrating its implementation. These analyses can be applied to the Extended Preorder Index and could be extended to investigate the specific properties of the PreLevel structure.

In [GST04], the authors describe a compiler that translates XQuery expressions into a relational algebra which may be efficiently implemented on top of any RDBMS. The translation procedure is fully compositional and emits algebraic code that strictly adheres to the XQuery language semantics of document and sequence order as well as node identity. These mappings turn RDBMSs into relational XML processors. Such XPath relational processors are the key to the seamless co-existence of SQL and XML data. The deployment of the PreLevel structure in the construction of a relational XPath and XQuery processor offers interesting challenges and avenues of research.

6.2.2 Support for Updates

The PreLevel structure is presented as a read-only index in this research thesis. However, the PreLevel structure may be adapted to incorporate insertions and deletions without a significant impact on query performance. In [GVT04], the author details several indexing strategies, such as B+tree and R-trees that facilitate updates with acceptable query performance. To facilitate the updating of the Level index, a relational storage structure called the Relational Interval tree (RI-tree), presented in [KPS00], is tailored to respond to updates and interval queries of the form [a,b]. The RI-tree may be investigated to determine its suitability as an dynamic index structure for our Level index.

However, one significant problem remains: the update of the *position* column within the Extended Preorder Index. Due to the properties of the *position* column and its intrinsic mapping to the Level index, support for updates in this column would significantly impact on the query performance of the PreLevel structure. To overcome this obstacle, the *position* column may be removed altogether. The impact on the query performance of the PreLevel structure by the removal of the *position* column to facilitate updates offers an interesting avenue of research.

Bibliography

[ABFS02] Bernd Amann, Catriel Beeri, Irini Fundulaki, and Michel Scholl. Querying XML Sources using an Ontology-based Mediator. In *Proceedings of the Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 429–448. LNCS 2519, Springer, 2002.

[AYCLS01] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of Tree Pattern Queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on the Management of Data*, pages 497–508. ACM Press, 2001.

[BBC+98] Jon Bosak, Tim Bray, Dan Connolly, Eve Maler, Gavin Nicol, Michael Sperberg-McQueen, Lauren Wood, and James Clark. W3C XML Specification DTD. Online Resource http://www.w3.org/XML/1998/06/xmlspec-report.htm, 1998.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, pages 310–321. ACM Press, 2002.

[Bou05] Ronald Bourret. XML and Databases. Online Resource http://www.rpbourret.com/xml/XMLAndDatabases.htm, 2005.

[CDF+04] Don Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jérôme Siméon, Jim Tivy, and Philip Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2004.

[CFZ04] Chee Yong Chan, Wenfei Fan, and Yiming Zeng. Taming XPath Queries by Minimizing Wildcard Steps. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, pages 156–167. Morgan Kaufmann, 2004.

[CRZ03] Akmal Chaudhri, Awais Rashid, and Roberto Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.

[CSF+01] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proceedings of 27th International Conference on Very Large Databases (VLDB)*, pages 341–350. Morgan Kaufmann, 2001.

[Die82] Paul F Dietz. Maintaining Order in a Linked List. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 122–127. ACM Press, 1982.

[Eur99] European Commission. The Bologna Declaration. Online Resource http://europa.eu.int/comm/education/policies/educ/bologna/bologna.pdf, 1999.

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, pages 95–106. Morgan Kaufmann, 2002.

[Gru02] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, pages 109–120. ACM Press, 2002.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, pages 252–263. Morgan Kaufmann, 2004.

[GVT04] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1): 91–131, 2004.

[Kay04] Michael Kay. *XPath 2.0 Programmer's Reference*. Wiley Publishing, 2004.

[KPS00] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing Intervals Efficiently in Object-Relational Databases. In *Proceedings of the 26th International Conference on Very Large DataBases (VLDB)*, pages 407–418. Morgan Kaufmann, 2000.

[KR05] Noel King and Mark Roantree. Process Composition Using a Semantic Registry. In *Proceedings of the CAISE'05 Workshops*, pages 271–285. Springer, 2005.

[LM01] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, pages 361–370. Morgan Kaufmann, 2001.

[LYYB96] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. Index Structures for Structured Documents. In *Proceedings of the 1st ACM International Conference on Digital Libraries*, pages 91–99. ACM Press, 1996.

[MAA+03] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Frederic Dang Ngoc. Exchanging Intensional XML Data. In *Proceedings of the 2003 ACM SIGMOD International Conference on the Management of Data*, pages 289–300. ACM Press, 2003.

[MBC+04] James McGovern, Per Bothner, Kurt Cagle, James Linn, and Vaidyanathan Nagarajan. *XQuery Kick Start*. Sams Publishing, 2004.

[MBV03] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML Web: A First Study. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, pages 500–510. ACM Press, 2003.

[Mei02] Wolfgang Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems, NODe Web and Database-Related Workshops*, pages 169–183. LNCS 2593, Springer, 2002.

[MP01] Peter McBrien and Alexandra Poulovassilis. A Semantic Approach to Integrating XML and Structured Data Sources. In *Proceedings of the 13th International Conference, CAiSE 2001, Advanced Information Systems Engineering*, pages 330–345. LNCS 2068, Springer, 2001.

[MS99] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295. LNCS 1540, Springer, 1999.

[MWA+98] Jason McHugh, Jennifer Widom, Serge Abiteboul, Qingshan Luo, and Anand Rajamaran. Indexing Semistructured Data. Technical Report, Computer Science Department, Stanford University. Online Resource http://citeseer.ist.psu.edu/mchugh98indexing.html, 1998.

[NLB+02] Ullas Nambiar, Zoé Lacroix, Stéphane Bressan, Mong-Li Lee, and Ying Guang Li. Efficient XML Data Management: An Analysis. In *Proceeding of the 3rd International Conference on E-Commerce and Web Technologies (EC-Web)*, pages 87–98. LNCS 2455, Springer, 2002.

[OBR05] Martin F O'Connor, Zohra Bellashène, and Mark Roantree. An Extended Preorder Index for Optimising XPath Expressions. In *Proceedings of the 3rd International XML Database Symposium (XSym 2005)*, pages 114–128. LNCS 3671, Springer, 2005.

[OOP+04] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on the Management of Data*, pages 903–908. ACM Press, 2004.

[Ora03] Oracle XML DB Developer's Guide 10g Release 1 (10.1). Online Resource http://otn.oracle.com, 2003.

[Ram02] Prakash Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, pages 299–309. ACM Press, 2002.

[SBKJ02] Marko Smiljani, Henk Blanken, Maurice Van Keulen, and Willem Jonker. Distributed XML Database Systems. Technical Report, Database group, Faculty of Informatics, Twente University. Online Resource http://doc.utwente.nl/fid/1166, 2002.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[SHYY05] Adam Silberstein, Hao He, Ke Yi, and Jun Yang. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *Proceedings of the 21th International Conference on Database Engineering*, pages 285–296. IEEE Computer Society, 2005.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML using a Relational Database System. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, pages 204–215. ACM Press, 2002.

[VFS04] Avinash Vyas, Mary F. Fernández, and Jérôme Siméon. The Simplest XML Storage Manager Ever. In *Proceedings of the 1st International Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/> in cooperation with ACM SIGMOD*, pages 37–42, 2004.

[Wor04a] World Wide Web Consortium. *Document Object Model (DOM) Level 3 Core Specification*, W3C Recommendation edition, April 2004.

[Wor04b] World Wide Web Consortium. *eXtensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation edition, February 2004.

[Wor04c] World Wide Web Consortium. *XML Schema Parts 0-2 [Primer, Structures, Datatypes] (Second Edition)*, W3C Recommendation edition, October 2004.

[Wor05a] World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, W3C Working Draft edition, April 2005.

[Wor05b] World Wide Web Consortium. *XML Path Language (XPath) 2.0*, W3C Working Draft edition, February 2005.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on the Management of Data*, pages 425–436. ACM Press, 2001.